Introduction
This collection of modules is from a Rice University, ECE Department Technical Report written around September 1994. It grew out of the doctoral and post doctoral research of Ivan Selesnick working with Prof. C. Sidney Burrus at Rice. Earlier reports on this work were published in the ICASSP and ISCAS conference proceedings in 1992-94 and a fairly complete report was published in the IEEE Transaction on Signal Processing in January 1996.

Introduction

The development of algorithms for the fast computation of the Discrete Fourier Transform in the last 30 years originated with the radix 2 Cooley-Tukey FFT and the theory and variety of FFTs has grown significantly since then. Most of the work has focused on FFTs whose sizes are composite, for the algorithms depend on the ability to factor the length of the data sequence, so that the transform can be found by taking the transform of smaller lengths. For this reason, algorithms for prime length transforms are building blocks for many composite length FFTs - the maximum length and the variety of lengths of a PFA or WFTA algorithm depend upon the availability of prime length FFT modules. As such, prime length Fast Fourier Transforms are a special, important and difficult case.

Fast algorithms designed for specific short prime lengths have been developed and have been written as straight line code [link], [link]. These dedicated programs rely upon an observation made in Rader's paper [link] in which he shows that a prime p length DFT can be found by performing a $p-1$ length circular convolution. Since the publication of that paper, Winograd had developed a theory of multiplicative complexity for transforms and designed algorithms for convolution that attain the minimum number of multiplications [link]. Although Winograd's algorithms are very efficient for small prime lengths, for longer lengths they require a large number of additions and the algorithms become very cumbersome to design. This has prevented the design of useful prime length FFT programs for lengths greater than 31. Although we have previously reported the design of programs for prime lengths greater than 31 [link] those programs required more additions than necessary and were long. Like the previously

existing ones, they simply consisted of a long list of instructions and did not take advantage of the attainable common structures.

In this paper we describe a set of programs for circular convolution and prime length FFTs that are are short, possess great structure, share many computational procedures, and cover a large variety of lengths. Because the underlying convolution is decomposed into a set of disjoint operations they can be performed in parallel and this parallelism is made clear in the programs. Moreover, each of these independent operations is made up of a sequence of sub-operations of the form $I \otimes A \otimes I$ where $\otimes$ denotes the Kronecker product. These operations can be implemented as vector/parallel operations [link]. Previous programs for prime length FFTs do not have these features: they consist of straight line code and are not amenable to vector/parallel implementations.

We have also developed a program that automatically generates these programs for circular convolution and prime length DFTs. This code generating program requires information only about a set of modules for computing cyclotomic convolutions. We compute these non-circular convolutions by computing a linear convolution and reducing the result. Furthermore, because these linear convolution algorithms can be built from smaller ones, the only modules needed are ones for the linear convolution of prime length sequences. It turns out that with linear convolution algorithms for only the lengths 2 and 3, we can generate a wide variety of prime length FFT algorithms. In addition, the code we generate is made up of calls to a relatively small set of functions. Accordingly, the subroutines can be designed and optimized to specifically suit a given architecture.

The programs we describe use Rader's conversion of a prime point DFT into a circular convolution, but this convolution we compute using the split nesting algorithm [link]. As Stasinski notes [link], this yields algorithms possessing greater structure and simpler programs and doesn't generally require more computation.

On the Row-Column Method

In computing the DFT of an $n = n_1 n_2$ point sequence where n_1 and n_2 are relatively prime, a row-column method can be employed. That is, if an $n_1 \times n_2$ array is appropriately formed from the n point sequence, then its DFT can be computed by computing the DFT of the rows and by then computing the DFT of the columns. The separability of the DFT makes this possible. It should be mentioned, however, that in at least two papers [link], [link] it is mistakenly assumed that the row-column method can also be applied to convolution. Unfortunately, the convolution of two sequences can not be found by forming two arrays, by convolving their rows, and by then convolving their columns. This misunderstanding about the separability of convolution also appears in [link] where the author incorrectly writes a diagonal matrix of a bilinear form as a Kronecker product. If it were a Kronecker product, then there would indeed exist a row-column method for convolution.

Earlier reports on this work were published in the conference proceedings [link], [link], [link] and a fairly complete report was published in the IEEE Transaction on Signal Processing [link]. Some parts of this approach appear in the Connexions book, Fast Fourier Transforms. This work is built on and an extension of that in [link] which is also in the Connexions Technical Report.

Preliminaries
This collection of modules is from a Rice University, ECE Department Technical Report written around September 1994. It grew out of the doctoral and post doctoral research of Ivan Selesnick working with Prof. C. Sidney Burrus at Rice. Earlier reports on this work were published in the ICASSP and ISCAS conference proceedings in 1992-94 and a fairly complete report was published in the IEEE Transaction on Signal Processing in January 1996.

Preliminaries

Because we compute prime point DFTs by converting them in to circular convolutions, most of this and the next section is devoted to an explanation of the split nesting convolution algorithm. In this section we introduce the various operations needed to carry out the split nesting algorithm. In particular, we describe the prime factor permutation that is used to convert a one-dimensional circular convolution into a multi-dimensional one. We also discuss the reduction operations needed when the Chinese Remainder Theorem for polynomials is used in the computation of convolution. The reduction operations needed for the split nesting algorithm are particularly well organized. We give an explicit matrix description of the reduction operations and give a program that implements the action of these reduction operations.

The presentation relies upon the notions of similarity transformations, companion matrices and Kronecker products. With them, we describe the split nesting algorithm in a manner that brings out its structure. We find that when companion matrices are used to describe convolution, the reduction operations block diagonalizes the circular shift matrix.

The companion matrix of a monic polynomial,
$M(s) = m_0 + m_1 s + \cdots + m_{n-1} s^{n-1} + s^n$ is given by
Equation:

$$C_M = \begin{bmatrix} & & -m_0 1 \\ 1 & & -m_1 \\ & \ddots & \vdots \\ & 1 & -m_{n-1} \end{bmatrix}.$$

Its usefulness in the following discussion comes from the following relation which permits a matrix formulation of convolution. Let

Equation:

$$\begin{aligned} X(s) &= x_0 + x_1 s + \cdots x_{n-1} s^{n-1} \\ H(s) &= h_0 + h_1 s + \cdots h_{n-1} s^{n-1} \\ Y(s) &= y_0 + y_1 s + \cdots y_{n-1} s^{n-1} \\ M(s) &= m_0 + m_1 s + \cdots m_{n-1} s^{n-1} + s^n \end{aligned}$$

Then

Equation:

$$Y(s) = \Big\langle H(s) X(s) \Big\rangle_{M(s)} \quad \Leftrightarrow \quad y = \left(\sum_{k=0}^{n-1} h_k C_M^k \right) x$$

where $y = (y_0, \cdots, y_{n-1})^t$, $x = (x_0, \cdots, x_{n-1})^t$, and C_M is the companion matrix of $M(s)$. In [link], we say y is the convolution of x and h with respect to $M(s)$. In the case of circular convolution, $M(s) = s^n - 1$ and C_{s^n-1} is the circular shift matrix denoted by S_n,

Equation:

$$S_n = \begin{bmatrix} & & 1 \\ 1 & & \\ & \ddots & \\ & 1 & \end{bmatrix}$$

Notice that any circulant matrix can be written as $\sum_k h_k S_n^k$.

Similarity transformations can be used to interpret the action of some convolution algorithms. If $C_M = T^{-1}AT$ for some matrix T (C_M and A are similar, denoted $C_M \sim A$), then [link] becomes
Equation:

$$y = T^{-1}\left(\sum_{k=0}^{n-1} h_k A^k\right) Tx.$$

That is, by employing the similarity transformation given by T in this way, the action of S_n^k is replaced by that of A^k. Many circular convolution algorithms can be understood, in part, by understanding the manipulations made to S_n and the resulting new matrix A. If the transformation T is to be useful, it must satisfy two requirements: (1) Tx must be simple to compute, and (2) A must have some advantageous structure. For example, by the convolution property of the DFT, the DFT matrix F diagonalizes S_n,
Equation:

$$S_n = F^{-1}\begin{bmatrix} w^0 & & & \\ & w^1 & & \\ & & \ddots & \\ & & & w^{n-1} \end{bmatrix} F$$

so that it diagonalizes every circulant matrix. In this case, Tx can be computed by using an FFT and the structure of A is the simplest possible. So the two above mentioned conditions are met.

The Winograd Structure can be described in this manner also. Suppose $M(s)$ can be factored as $M(s) = M_1(s)M_2(s)$ where M_1 and M_2 have no common roots, then $C_M \sim (C_{M_1} \oplus C_{M_2})$ where $\oplus$ denotes the matrix direct sum. Using this similarity and recalling [link], the original convolution is decomposed into disjoint convolutions. This is, in fact, a statement of the Chinese Remainder Theorem for polynomials expressed in matrix notation.

In the case of circular convolution, $s^n - 1 = \prod_{d|n} \Phi_d(s)$, so that S_n can be transformed to a block diagonal matrix,
Equation:

$$S_n \sim \begin{bmatrix} C_{\Phi_1} & & & \\ & C_{\Phi_d} & & \\ & & \ddots & \\ & & & C_{\Phi_n} \end{bmatrix} = \left(\bigoplus_{d|n} C_{\Phi_d} \right)$$

where $\Phi_d(s)$ is the d^{th} cyclotomic polynomial. In this case, each block represents a convolution with respect to a cyclotomic polynomial, or a `cyclotomic convolution'. Winograd's approach carries out these cyclotomic convolutions using the Toom-Cook algorithm. Note that for each divisor, d, of n there is a corresponding block on the diagonal of size $\varphi(d)$, for the degree of $\Phi_d(s)$ is $\varphi(d)$ where $\varphi(\cdot)$ is the Euler totient function. This method is good for short lengths, but as n increases the cyclotomic convolutions become cumbersome, for as the number of distinct prime divisors of d increases, the operation described by $\sum_k h_k (C_{\Phi_d})^k$ becomes more difficult to implement.

The Agarwal-Cooley Algorithm utilizes the fact that
Equation:

$$S_n = P^t (S_{n_1} \otimes S_{n_2}) P$$

where $n = n_1 n_2$, $(n_1, n_2) = 1$ and P is an appropriate permutation [link]. This converts the one dimensional circular convolution of length n to a two dimensional one of length n_1 along one dimension and length n_2 along the second. Then an n_1-point and an n_2-point circular convolution algorithm can be combined to obtain an n-point algorithm. In polynomial notation, the mapping accomplished by this permutation P can be informally indicated by
Equation:

$$Y(s) = \left\langle X(s)H(s) \right\rangle_{s^n-1} \overset{P}{\Leftrightarrow} Y(s,t) = \left\langle X(s,t)H(s,t) \right\rangle_{s^{n_1}-1,t^{n_2}-1}.$$

It should be noted that [link] implies that a circulant matrix of size $n_1 n_2$ can be written as a block circulant matrix with circulant blocks.

The Split-Nesting algorithm [link] combines the structures of the Winograd and Agarwal-Cooley methods, so that S_n is transformed to a block diagonal matrix as in [link],
Equation:

$$S_n \sim \bigoplus_{d|n} \Psi(d).$$

Here $\Psi(d) = \otimes_{p|d, p\in\mathscr{P}} C_{\Phi_{H_d(p)}}$ where $H_d(p)$ is the highest power of p dividing d, and $\mathscr{P}$ is the set of primes.

Example:
Equation:

$$S_{45} \sim \begin{bmatrix} 1 & & & & & & \\ & C_{\Phi_3} & & & & & \\ & & C_{\Phi_9} & & & & \\ & & & C_{\Phi_5} & & & \\ & & & & C_{\Phi_3}\otimes C_{\Phi_5} & & \\ & & & & & C_{\Phi_9}\otimes C_{\Phi_5} \end{bmatrix}$$

In this structure a multidimensional cyclotomic convolution, represented by $\Psi(d)$, replaces each cyclotomic convolution in Winograd's algorithm (represented by C_{Φ_d} in [link]. Indeed, if the product of $b_1, \cdots, b_k$ is d and

they are pairwise relatively prime, then $C_{\Phi_d} \sim C_{\Phi_{b_1}} \otimes \cdots \otimes C_{\Phi_{b_k}}$. This gives a method for combining cyclotomic convolutions to compute a longer circular convolution. It is like the Agarwal-Cooley method but requires fewer additions [link].

Prime Factor Permutations

One can obtain $S_{n_1} \otimes S_{n_2}$ from $S_{n_1 n_2}$ when $(n_1, n_2) = 1$, for in this case, S_n is similar to $S_{n_1} \otimes S_{n_2}$, $n = n_1 n_2$. Moreover, they are related by a permutation. This permutation is that of the prime factor FFT algorithms and is employed in nesting algorithms for circular convolution [link], [link]. The permutation is described by Zalcstein [link], among others, and it is his description we draw on in the following.

Let $n = n_1 n_2$ where $(n_1, n_2) = 1$. Define e_k, $(0 \leq k \leq n-1)$, to be the standard basis vector, $(0, \cdots, 0, 1, 0, \cdots, 0)^t$, where the 1 is in the k^{th} position. Then, the circular shift matrix, S_n, can be described by
Equation:

$$S_n e_k = e_{\langle k+1 \rangle_n}.$$

Note that, by inspection,
Equation:

$$(S_{n_2} \otimes S_{n_1}) e_{a+n_1 b} = e_{\langle a+1 \rangle_{n_1} + n_1 \langle b+1 \rangle_{n_2}}$$

where $0 \leq a \leq n_1 - 1$ and $0 \leq b \leq n_2 - 1$. Because n_1 and n_2 are relatively prime a permutation matrix P can be defined by
Equation:

$$P e_k = e_{\langle k \rangle_{n_1} + n_1 \langle k \rangle_{n_2}}.$$

With this P,
Equation:

$$\begin{aligned} PS_n e_k &= Pe_{\langle k+1\rangle_n} \\ &= e_{\langle \langle k+1\rangle_n\rangle_{n_1}+n_1\langle \langle k+1\rangle_n\rangle_{n_2}} \\ &= e_{\langle k+1\rangle_{n_1}+n_1\langle k+1\rangle_{n_2}} \end{aligned}$$

and
Equation:

$$\begin{aligned} (S_{n_2}\otimes S_{n_1})Pe_k &= (S_{n_2}\otimes S_{n_1})e_{\langle k\rangle_{n_1}+n_1\langle k\rangle_{n_2}} \\ &= e_{\langle k+1\rangle_{n_1}+n_1\langle k+1\rangle_{n_2}}. \end{aligned}$$

Since $PS_n e_k = (S_{n_2}\otimes S_{n_1})Pe_k$ and $P^{-1}=P^t$, one gets, in the multi-factor case, the following.
Lemma

If $n = n_1\cdots n_k$ and $n_1, ..., n_k$ are pairwise relatively prime, then $S_n = P^t\,(S_{n_k}\otimes\cdots\otimes S_{n_1})P$ where P is the permutation matrix given by $Pe_k = e_{\langle k\rangle_{n_1}+n_1\langle k\rangle_{n_2}+\cdots+n_1\cdots n_{k-1}\langle k\rangle_{n_k}}$.

This useful permutation will be denoted here as $P_{n_k,\cdots,n_1}$. If $n = p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k}$ then this permutation yields the matrix, $S_{p_1^{e_1}}\otimes\cdots\otimes S_{p_k^{e_k}}$. This product can be written simply as $\bigotimes_{i=1}^{k} S_{p_i^{e_i}}$, so that one has

$$S_n = P^t_{n_1,\cdots,n_k}\left(\bigotimes_{i=1}^{k} S_{p_i^{e_i}}\right)P_{n_1,\cdots,n_k}.$$

It is quite simple to show that
Equation:

$$P_{a,b,c} = (I_a\otimes P_{b,c})P_{a,bc} = (P_{a,b}\otimes I_c)P_{ab,c}.$$

In general, one has
Equation:

$$P_{n_1,\cdots,n_k} = \prod_{i=2}^{k} (P_{n_1 \cdots n_{i-1},n_i} \otimes I_{n_{i+1} \cdots n_k}).$$

A Matlab function for $P_{a,b} \otimes I_s$ is `pfp2I()` in one of the appendices. This program is a direct implementation of the definition. In a paper by Templeton [link], another method for implementing $P_{a,b}$, without `if' statements, is given. That method requires some precalculations, however. A function for $P_{n_1,\cdots,n_k}$ is `pfp()`. It uses [link] and calls `pfp2I()` with the appropriate arguments.

Reduction Operations

The Chinese Remainder Theorem for polynomials can be used to decompose a convolution of two sequences (the polynomial product of two polynomials evaluated modulo a third polynomial) into smaller convolutions (smaller polynomial products) [link]. The Winograd n point circular convolution algorithm requires that polynomials are reduced modulo the cyclotomic polynomial factors of $s^n - 1$, $\Phi_d(s)$ for each d dividing n.

When n has several prime divisors the reduction operations become quite complicated and writing a program to implement them is difficult. However, when n is a prime power, the reduction operations are very structured and can be done in a straightforward manner. Therefore, by converting a one-dimensional convolution to a multi-dimensional one, in which the length is a prime power along each dimension, the split nesting algorithm avoids the need for complicated reductions operations. This is one advantage the split nesting algorithm has over the Winograd algorithm.

By applying the reduction operations appropriately to the circular shift matrix, we are able to obtain a block diagonal form, just as in the Winograd convolution algorithm. However, in the split nesting algorithm, each diagonal block represents multi-dimensional cyclotomic convolution rather than a one-dimensional one. By forming multi-dimensional convolutions out of one-dimensional ones, it is possible to combine algorithms for smaller convolutions (see the next section). This is a second advantage split nesting

algorithm has over the Winograd algorithm. The split nesting algorithm, however, generally uses more than the minimum number of multiplications.

Below we give an explicit matrix description of the required reduction operations, give a program that implements them, and give a formula for the number of additions required. (No multiplications are needed.)

First, consider $n = p$, a prime. Let
Equation:

$$X(s) = x_0 + x_1 s + \cdots + x_{p-1} s^{p-1}$$

and recall $s^p - 1 = (s-1)\left(s^{p-1} + s^{p-2} + \cdots + s + 1\right) = \Phi_1(s)\Phi_p(s)$. The residue $\langle X(s) \rangle_{\Phi_1(s)}$ is found by summing the coefficients of $X(s)$. The residue $\langle X(s) \rangle_{\Phi_p(s)}$ is given by $\sum_{k=0}^{p-2} (x_k - x_{p-1}) s^k$. Define R_p to be the matrix that reduces $X(s)$ modulo $\Phi_1(s)$ and $\Phi_p(s)$, such that if $X_0(s) = \langle X(s) \rangle_{\Phi_1(s)}$ and $X_1(s) = \langle X(s) \rangle_{\Phi_p(s)}$ then
Equation:

$$\begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = R_p X$$

where X, X_0 and X_1 are vectors formed from the coefficients of $X(s)$, $X_0(s)$ and $X_1(s)$. That is,
Equation:

$$R_p = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & & & & -1 \\ & 1 & & & -1 \\ & & 1 & & -1 \\ & & & 1 & -1 \end{bmatrix}$$

so that $R_p = \begin{bmatrix} 1_{-1} \\ G_p \end{bmatrix}$ where G_p is the $\Phi_p(s)$ reduction matrix of size $(p-1) \times p$. Similarly, let $X(s) = x_0 + x_1 s + \cdots + x_{p^{e-1}} s^{p^{e-1}}$ and define R_{p^e} to be the matrix that reduces $X(s)$ modulo $\Phi_1(s)$, $\Phi_p(s)$, ..., $\Phi_{p^e}(s)$ such that

Equation:

$$\begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_e \end{bmatrix} = R_{p^e} X,$$

where as above, X, X_0, ..., X_e are the coefficients of $X(s)$, $\langle X(s) \rangle_{\Phi_1(s)}$, ..., $\langle X(s) \rangle_{\Phi_{p^e}(s)}$.

It turns out that R_{p^e} can be written in terms of R_p. Consider the reduction of $X(s) = x_0 + \cdots + x_8 s^8$ by $\Phi_1(s) = s - 1$, $\Phi_3(s) = s^2 + s + 1$, and $\Phi_9(s) = s^6 + s^3 + 1$. This is most efficiently performed by reducing $X(s)$ in two steps. That is, calculate $X'(s) = \langle X(s) \rangle_{s^3 - 1}$ and $X_2(s) = \langle X(s) \rangle_{s^6 + s^3 + 1}$. Then calculate $X_0(s) = \langle X'(s) \rangle_{s-1}$ and $X_1(s) = \langle X'(s) \rangle_{s^2+s+1}$. In matrix notation this becomes

Equation:

$$\begin{bmatrix} X' \\ X_2 \end{bmatrix} = \begin{bmatrix} I_3 & I_3 & I_3 \\ I_3 & & -I_3 \\ & I_3 & -I_3 \end{bmatrix} X \quad \text{and} \quad \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & & -1 \\ & 1 & -1 \end{bmatrix} X'.$$

Together these become

Equation:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} R_3 & & \\ & I_3 & \\ & & I_3 \end{bmatrix} \begin{bmatrix} I_3 & I_3 & I_3 \\ I_3 & & -I_3 \\ & I_3 & -I_3 \end{bmatrix} X$$

or
Equation:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = (R_3 \oplus I_6)\,(R_3 \otimes I_3) X$$

so that $R_9 = (R_3 \oplus I_6)\,(R_3 \otimes I_3)$ where $\oplus$ denotes the matrix direct sum. Similarly, one finds that $R_{27} = (R_3 \oplus I_{24})\,((R_3 \otimes I_3) \oplus I_{18})\,(R_3 \otimes I_9)$. In general, one has the following.
Lemma

R_{p^e} is a $p^e \times p^e$ matrix given by $R_{p^e} = \prod_{k=0}^{e-1} \left(\left(R_p \otimes I_{p^k}\right) \oplus I_{p^e - p^{k+1}}\right)$ and can be implemented with $2(p^e - 1)$ additions.

The following formula gives the decomposition of a circular convolution into disjoint non-circular convolutions when the number of points is a prime power.
Equation:

$$\begin{aligned} R_{p^e}\, S_{p^e}\, R_{p^e}^{-1} &= \begin{bmatrix} 1 & & & \\ & C_{\Phi_p} & & \\ & & \ddots & \\ & & & C_{\Phi_{p^e}} \end{bmatrix} \\ &= \bigoplus_{i=0}^{e} C_{\Phi_{p^i}} \end{aligned}$$

Example:
Equation:

$$R_9\, S_9\, R_9^{-1} = \begin{bmatrix} 1 & & \\ & C_{\Phi_3} & \\ & & C_{\Phi_9} \end{bmatrix}$$

It turns out that when n is not a prime power, the reduction of polynomials modulo the cyclotomic polynomial $\Phi_n(s)$ becomes complicated, and with an increasing number of prime factors, the complication increases. Recall, however, that a circular convolution of length $p_1^{e_1} \cdots p_k^{e_k}$ can be converted (by an appropriate permutation) into a k dimensional circular convolution of length $p_i^{e_i}$ along dimension i. By employing this one-dimensional to multi-dimensional mapping technique, one can avoid having to perform polynomial reductions modulo $\Phi_n(s)$ for non-prime-power n.

The prime factor permutation discussed previously is the permutation that converts a one-dimensional circular convolution into a multi-dimensional one. Again, we can use the Kronecker product to represent this. In this case, the reduction operations are applied to each matrix in the following way:
Equation:

$$T\left(S_{p_1^{e_1}} \otimes \cdots \otimes S_{p_k^{e_k}}\right)T^{-1} = \left(\oplus_{i=0}^{e_1} C_{\Phi_{p_1^i}}\right) \otimes \cdots \otimes \left(\oplus_{i=0}^{e_k} C_{\Phi_{p_k^i}}\right)$$

where
Equation:

$$T = R_{p_1^{e_1}} \otimes \cdots \otimes R_{p_k^{e_k}}$$

Example:
Equation:

$$T(S_9 \otimes S_5)T^{-1} = \begin{bmatrix} 1 & & \\ & C_{\Phi_3} & \\ & & C_{\Phi_9} \end{bmatrix} \otimes \begin{bmatrix} 1 & \\ & C_{\Phi_5} \end{bmatrix}$$

where $T = R_9 \otimes R_5$.

The matrix $R_{p_1^{e_1}} \otimes \cdots \otimes R_{p_k^{e_k}}$ can be factored using a property of the Kronecker product. Notice that $(A \otimes B) = (A \otimes I)(I \otimes B)$, and $(A \otimes B \otimes C) = (A \otimes I)(I \otimes B \otimes I)(I \otimes C)$ (with appropriate dimensions) so that one gets

Equation:

$$\bigotimes_{i=1}^{k} R_{p_i^{e_i}} = \prod_{i=1}^{k} \left(I_{m_i} \otimes R_{p_i^{e_i}} \otimes I_{n_i} \right),$$

where $m_i = \prod_{j=1}^{i-1} p_j^{e_j}$, $n_i = \prod_{j=i+1}^{k} p_j^{e_j}$ and where the empty product is taken to be 1. This factorization shows that T can be implemented basically by implementing copies of R_{p^e}. There are many variations on this factorization as explained in [link]. That the various factorization can be interpreted as vector or parallel implementations is also explained in [link].

Example:

Equation:

$$R_9 \otimes R_5 = (R_9 \otimes I_5)\,(I_9 \otimes R_5)$$

and

Equation:

$$R_9 \otimes R_{25} \otimes R_7 = (R_9 \otimes I_{175})\,(I_9 \otimes R_{25} \otimes I_7)\,(I_{225} \otimes R_7)$$

When this factored form of $\otimes R_{n_i}$ or any of the variations alluded to above, is used, the number of additions incurred is given by
Equation:

$$\begin{aligned}\sum_{i=1}^{k} \frac{N}{p_i^{e_i}} \mathscr{A}\left(R_{p_i^{e_i}}\right) &= \sum_{i=1}^{k} \frac{N}{p_i^{e_i}} 2\left(p_i^{e_i}-1\right) \\ &= 2N \sum_{i=1}^{k} 1-\frac{1}{p_i^{e_i}} \\ &= 2N\left(k-\sum_{i=1}^{k} \frac{1}{p_i^{e_i}}\right)\end{aligned}$$

where $N=p_1^{e_1} \cdots p_k^{e_k}$.

Although the use of operations of the form $R_{p_1^{e_1}} \otimes \cdots \otimes R_{p_k^{e_k}}$ is simple, it does not exactly separate the circular convolution into smaller disjoint convolutions. In other words, its use does not give rise in [link] and [link] to block diagonal matrices whose diagonal blocks are the form $\otimes_i C_{\Phi_{p_i}}$. However, by reorganizing the arrangement of the operations we can obtain the block diagonal form we seek.

First, suppose A, B and C are matrices of sizes $a \times a$, $b \times b$ and $c \times c$ respectively. If
Equation:

$$TBT^{-1}=\begin{bmatrix} B_1 & \\ & B_2 \end{bmatrix}$$

where B_1 and B_2 are matrices of sizes $b_1 \times b_1$ and $b_2 \times b_2$, then
Equation:

$$Q\left(A \otimes B \otimes C\right)Q^{-1}=\begin{bmatrix} A \otimes B_1 \otimes C & \\ & A \otimes B_2 \otimes C \end{bmatrix}$$

where
Equation:

$$Q = \begin{bmatrix} I_a \otimes T\left(1 : b_1, :\right) \otimes I_c \\ I_a \otimes T\left(b_1 + 1 : b, :\right) \otimes I_c \end{bmatrix}.$$

Here $T(1 : b_1, :)$ denotes the first b_1 rows and all the columns of T and similarly for $T(b_1 + 1 : b, :)$. Note that
Equation:

$$\begin{bmatrix} A \otimes B_1 \otimes C & \\ & A \otimes B_2 \otimes C \end{bmatrix} \neq A \otimes \begin{bmatrix} B_1 & \\ & B_2 \end{bmatrix} \otimes C.$$

That these two expressions are not equal explains why the arrangement of operations must be reorganized in order to obtain the desired block diagonal form. The appropriate reorganization is described by the expression in [link]. Therefore, we must modify the transformation of [link] appropriately. It should be noted that this reorganization of operations does not change their computational cost. It is still given by [link].

For example, we can use this observation and the expression in [link] to arrive at the following similarity transformation:
Equation:

$$Q(S_{p_1} \otimes S_{p_2})Q^{-1} = \begin{bmatrix} 1 & & & \\ & C_{\Phi_{p_1}} & & \\ & & C_{\Phi_{p_2}} & \\ & & & C_{\Phi_{p_1}} \otimes C_{\Phi_{p_2}} \end{bmatrix}$$

where
Equation:

$$Q = \begin{bmatrix} I_{p_1} \otimes 1_{\text{-p}_2}^t \\ I_{p_1} \otimes G_{p_2} \end{bmatrix} (R_{p_1} \otimes I_{p_2})$$

$1_{\text{-p}}$ is a column vector of p 1's
Equation:

$$1_{\text{-p}} = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}^t$$

and G_p is the $(p-1) \times p$ matrix:
Equation:

$$G_p = \begin{bmatrix} 1 & & & -1 \\ & 1 & & -1 \\ & & \ddots & \vdots \\ & & 1 & -1 \end{bmatrix} = \left[I_{p-1} - \underline{1}_{p-1} \right].$$

In general we have
Equation:

$$R\left(S_{p_1^{e_1}} \otimes \cdots \otimes S_{p_k^{e_k}} \right) R^{-1} = \underset{d|n}{\oplus} \Psi\left(d\right)$$

where $R = R_{p_1^{e_1}, \cdots, p_k^{e_k}}$ is given by
Equation:

$$R_{p_1^{e_1}, \cdots, p_k^{e_k}} = \prod_{i=k}^{1} Q\left(m_i, p_i^{e_i}, n_i\right)$$

with $m_i = \prod_{j=1}^{i-1} p_j^{e_j}$, $n_i = \prod_{j=i+1}^{k} p_j^{e_j}$ and
Equation:

$$Q\left(a,p^e,c\right)=\prod_{j=0}^{e-1}\begin{bmatrix} I_a\otimes 1^t_{-p}\otimes I_{cp^j} & & \\ I_a\otimes G_p\otimes I_{cp^j} & & \\ & & I_{ac(p^e-p^{j+1})}\end{bmatrix}.$$

1_{-p} and G_p are as given in [link] and [link].

Example:
Equation:

$$R(S_9\otimes S_5)R^{-1}=\begin{bmatrix}1 & & & & & & & & \\ & C_{\Phi_3} & & & & & & & \\ & & & C_{\Phi_9} & & & & & \\ & & & & & C_{\Phi_5} & & & \\ & & & & & & C_{\Phi_3}\otimes C_{\Phi_5} & & \\ & & & & & & & & C_{\Phi_9}\otimes C_{\Phi_5}\end{bmatrix}$$

where
Equation:

$$\begin{aligned} R &= R_{9,5} \\ &= Q(9,5,1)Q(1,9,5)\end{aligned}$$

and R can be implemented with 152 additions.

Notice the distinction between this example and example "Reduction Operations". In example "Reduction Operations" we obtained from $S_9\otimes S_5$ a Kronecker product of two block diagonal matrices, but here we obtained a block diagonal matrix whose diagonal blocks are the Kronecker product of cyclotomic companion matrices. Each block in [link] represents a multi-dimensional cyclotomic convolution.

A Matlab program that carries out the operation $R_{p_1^{e_1},\ldots,p_k^{e_k}}$ in [link] is `KRED()`.

```
function x = KRED(P,E,K,x)
% x = KRED(P,E,K,x);
% P : P = [P(1),...,P(K)];
% E : E = [E(K),...,E(K)];
for i = 1:K
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
   e = E(i);
   for j = e-1:-1:0
      x(1:a*c*(p^(j+1))) = RED(p,a,c*(p^j),x(1:a*c*
(p^(j+1))));
   end
end
```

It calls the Matlab program `RED()`.

```
function y = RED(p,a,c,x)
% y = RED(p,a,c,x);
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      y(i+j+1) = x(i*p+j+1);
      for k = 0:c:c*(p-2)
         y(i+j+1) = y(i+j+1) + x(i*p+j+k+c+1);
         y(i*(p-1)+j+k+a*c+1) = x(i*p+j+k+1) -
x(i*p+j+c*(p-1)+1);
      end
   end
end
```

These two Matlab programs are not written to run as fast as they could be. They are a `naive' coding of $R_{p_1^{e_1},\ldots,p_k^{e_k}}$ and are meant to serve as a basis for more efficient programs. In particular, the indexing and the loop counters can be modified to improve the efficiency. However, the modifications that

minimize the overhead incurred by indexing operations depends on the programming language, the compiler and the computer used. These two programs are written with simple loop counters and complicated indexing operations so that appropriate modifications can be easily made.

Inverses

The inverse of R_p has the form
Equation:

$$R_p^{-1} = \frac{1}{p}\begin{bmatrix} 1 & p-1 & -1 & -1 & -1 \\ 1 & -1 & p-1 & -1 & -1 \\ 1 & -1 & -1 & p-1 & -1 \\ 1 & -1 & -1 & -1 & p-1 \\ 1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

and
Equation:

$$R_{p^e}^{-1} = \prod_{k=0}^{e-1} \left(\left(R_p^{-1} \otimes I_{p^{e-1-k}} \right) \oplus I_{p^e - p^{e-k}} \right).$$

Because the inverse of Q in [link] is given by
Equation:

$$Q^{-1} = \left[I_a \otimes T^{-1}\left(:, 1:b_1\right) \otimes I_c \quad I_a \otimes T^{-1}\left(:, b_1+1:b\right) \otimes I_c \right]$$

the inverse of the matrix R described by eqs [link], [link] and [link] is given by
Equation:

$$R^{-1} = \prod_{i=1}^{k} Q(m_i, p_i^{e_i}, n_i)^{-1}$$

with $m_i = \prod_{j=1}^{i-1} p_j^{e_j}$, $n_i = \prod_{j=i+1}^{k} p_j^{e_j}$ and
Equation:

$$Q(a, p^e, c)^{-1} = \prod_{j=e-1}^{0} \begin{bmatrix} I_a \otimes 1_{\text{-p}}^t \otimes I_{cp^j} & I_a \otimes V_p \otimes I_{cp^j} & \\ & & I_{ac(p^e - p^{j+1})} \end{bmatrix}$$

where V_p denotes the matrix in [link] without the first column. A Matlab program for R^t is `tKRED()` , it calls the Matlab program `tRED()` . A Matlab program for R^{-t} is `itKRED()` , it calls the Matlab program `itRED()` . These programs all appear in one of the appendices.

Bilinear Forms for Circular Convolution
This collection of modules is from a Rice University, ECE Department Technical Report written around September 1994. It grew out of the doctoral and post doctoral research of Ivan Selesnick working with Prof. C. Sidney Burrus at Rice. Earlier reports on this work were published in the ICASSP and ISCAS conference proceedings in 1992-94 and a fairly complete report was published in the IEEE Transaction on Signal Processing in January 1996.

Bilinear Forms for Circular Convolution

A basic technique in fast algorithms for convolution is that of interpolation. That is, two polynomials are evaluated at some common points and these values are multiplied [link], [link], [link]. By interpolating these products, the product of the two original polynomials can be determined. In the Winograd short convolution algorithms, this technique is used and the common points of evaluation are the simple integers, 0, 1, and -1. Indeed, the computational savings of the interpolation technique depends on the use of special points at which to interpolate. In the Winograd algorithm the computational savings come from the simplicity of the small integers. (As an algorithm for convolution, the FFT interpolates over the roots of unity.) This interpolation method is often called the Toom-Cook method and it is given by two matrices that describe a bilinear form.

We use bilinear forms to give a matrix formulation of the split nesting algorithm. The split nesting algorithm combines smaller convolution algorithms to obtain algorithms for longer lengths. We use the Kronecker product to explicitly describe the way in which smaller convolution algorithms are appropriately combined.

The Scalar Toom-Cook Method

First we consider the linear convolution of two n point sequences. Recall that the linear convolution of h and x can be represented by a matrix vector product. When $n = 3$:

Equation:

$$\begin{bmatrix} h_0 & & \\ h_1 & h_0 & \\ h_2 & h_1 & h_0 \\ & h_2 & h_1 \\ & & h_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

This linear convolution matrix can be written as $h_0 H_0 + h_1 H_1 + h_2 H_2$ where H_k are clear.

The product $\sum_{k=0}^{n-1} h_k H_k x$ can be found using the Toom-Cook algorithm, an interpolation method. Choose $2n-1$ interpolation points, $i_1, \cdots, i_{2n-1}$, and let A and C be matrices given by
Equation:

$$A = \begin{bmatrix} i_1^0 & \cdots & i_1^{n-1} \\ & \vdots & \\ i_{2n-1}^0 & \cdots & i_{2n-1}^{n-1} \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} i_1^0 & \cdots & i_1^{2n-2} \\ & \vdots & \\ i_{2n-1}^0 & \cdots & i_{2n-1}^{2n-2} \end{bmatrix}^{-1}.$$

That is, A is a degree $n-1$ Vandermonde matrix and C is the inverse of the degree $2n-2$ Vandermonde matrix specified by the same points specifying A. With these matrices, one has
Equation:

$$\sum_{k=0}^{n-1} h_k H_k x = C\{Ah^* Ax\}$$

where * denotes point by point multiplication. The terms Ah and Ax are the values of $H(s)$ and $X(s)$ at the points $i_1, \cdots i_{2n-1}$. The point by point

multiplication gives the values $Y(i_1), \cdots, Y(i_{2n-1})$. The operation of C obtains the coefficients of $Y(s)$ from its values at these points of evaluation. This is the bilinear form and it implies that
Equation:

$$H_k = \text{Cdiag}\,(Ae_k)A.$$

Example:
If $n = 2$, then equations [link] and [link] give
Equation:

$$\begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} x = C\{Ah^* Ax\}$$

When the interpolation points are 0, 1,and -1,
Equation:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & .5 & -.5 \\ -1 & .5 & .5 \end{bmatrix}$$

However, A and C do not need to be Vandermonde matrices as in [link]. For example, see the two point linear convolution algorithm in the appendix. As long as A and C are matrices such that $H_k = \text{Cdiag}\,(Ae_k)A$, then the linear convolution of x and h is given by the bilinear form $y = C\{Ah^* Ax\}$. More generally, as long as A, B and C are matrices satisfying $H_k = \text{Cdiag}\,(Be_k)A$, then $y = C\{Bh^* Ax\}$ computes the linear convolution of h and x. For convenience, if $C\{Ah^* Ax\}$ computes the n point linear convolution of h and x (both h and x are n point

sequences), then we say "(A, B, C) describes a bilinear form for n point linear convolution."

Similarly, we can write a bilinear form for cyclotomic convolution. Let d be any positive integer and let $X(s)$ and $H(s)$ be polynomials of degree $\varphi(d) - 1$ where $\varphi(\cdot)$ is the Euler totient function. If A, B and C are matrices satisfying $\left(C_{\Phi_d}\right)^k = \text{Cdiag}\left(Be_k\right)A$ for $0 \leq k \leq \varphi(d) - 1$, then the coefficients of $Y\left(s\right) = \left\langle X\left(s\right)H\left(s\right)\right\rangle_{\Phi_d(s)}$ are given by $y = C\{Bh^{*}Ax\}$. As above, if $y = C\{Bh^{*}Ax\}$ computes the d-cyclotomic convolution, then we say "(A, B, C) describes a bilinear form for $\Phi_d\left(s\right)$ convolution."

But since $\left\langle X\left(s\right)H\left(s\right)\right\rangle_{\Phi_d(s)}$ can be found by computing the product of $X(s)$ and $H(s)$ and reducing the result, a cyclotomic convolution algorithm can always be derived by following a linear convolution algorithm by the appropriate reduction operation: If G is the appropriate reduction matrix and if (A, B, F) describes a bilinear form for a $\varphi(d)$ point **linear** convolution, then (A, B, GF) describes a bilinear form for $\Phi_d\left(s\right)$ convolution. That is, $y = GF\{Bh^{*}Ax\}$ computes the coefficients of $\left\langle X\left(s\right)H\left(s\right)\right\rangle_{\Phi_d(s)}$.

Circular Convolution

By using the Chinese Remainder Theorem for polynomials, circular convolution can be decomposed into disjoint cyclotomic convolutions. Let p be a prime and consider p point circular convolution. Above we found that

Equation:

$$S_p = R_p^{-1}\begin{bmatrix} 1 & \\ & C_{\Phi_p} \end{bmatrix} R_p$$

and therefore

Equation:

$$S_p^k = R_p^{-1} \begin{bmatrix} 1 & \\ & C_{\Phi_p}^k \end{bmatrix} R_p.$$

If (A_p, B_p, C_p) describes a bilinear form for $\Phi_p(s)$ convolution, then
Equation:

$$S_p^k = R_p^{-1} \begin{bmatrix} 1 & \\ & C_p \end{bmatrix} \operatorname{diag}\left(\begin{bmatrix} 1 & \\ & B_p \end{bmatrix} R_p e_k \right) \begin{bmatrix} 1 & \\ & A_p \end{bmatrix} R_p$$

and consequently the circular convolution of h and x can be computed by
Equation:

$$y = R_p^{-1} C \left\{ B R_p h * A R_p x \right\}$$

where $A = 1 \oplus A_p$, $B = 1 \oplus B_p$ and $C = 1 \oplus C_p$. We say (A, B, C) describes a bilinear form for p point circular convolution. Note that if (D, E, F) describes a $(p-1)$ point linear convolution then A_p, B_p and C_p can be taken to be $A_p = D$, $B_p = E$ and $C_p = G_p F$ where G_p represents the appropriate reduction operations. Specifically, G_p is given by Equation 42 from Preliminaries.

Next we consider p^e point circular convolution. Recall that $S_{p^e} = R_{p^e}^{-1} \left(\oplus_{i=0}^{e} C_{\Phi_{p^i}} \right) R_{p^e}$ as in Equation 27 from Preliminaries so that the circular convolution is decomposed into a set of $e+1$ disjoint $\Phi_{p^i}(s)$ convolutions. If $\left(A_{p^i}, B_{p^i}, C_{p^i}\right)$ describes a bilinear form for $\Phi_{p^i}(s)$ convolution and if
Equation:

$$\begin{aligned} A &= 1 \oplus A_p \oplus \cdots \oplus A_{p^e} \\ B &= 1 \oplus B_p \oplus \cdots \oplus B_{p^e} \\ C &= 1 \oplus C_p \oplus \cdots \oplus C_{p^e} \end{aligned}$$

then $\left(AR_{p^e}, BR_{p^e}, R_{p^e}^{-1}C\right)$ describes a bilinear form for p^e point circular convolution. In particular, if (D_d, E_d, F_d) describes a bilinear form for d point linear convolution, then A_{p^i}, B_{p^i} and C_{p^i} can be taken to be
Equation:

$$\begin{aligned} A_{p^i} &= D_{\varphi(p^i)} \\ B_{p^i} &= E_{\varphi(p^i)} \\ C_{p^i} &= G_{p^i} F_{\varphi(p^i)} \end{aligned}$$

where G_{p^i} represents the appropriate reduction operation and $\varphi(\cdot)$ is the Euler totient function. Specifically, G_{p^i} has the following form
Equation:

$$G_{p^i} = \begin{bmatrix} I_{(p-1)p^{i-1}} & -1_{\text{-p-1}} \otimes I_{p^{i-1}} & \begin{bmatrix} I_{(p-2)p^{i-1}-1} \\ 0_{p^{i-1}+1,(p-2)p^{i-1}-1} \end{bmatrix} \end{bmatrix}$$

if $p \geq 3$, while
Equation:

$$G_{2^i} = \begin{bmatrix} I_{2^{i-1}} & \begin{bmatrix} -I_{2^{i-1}-1} \\ 0_{1,2^{i-1}-1} \end{bmatrix} \end{bmatrix}.$$

Note that the matrix R_{p^e} block diagonalizes S_{p^e} and each diagonal block represents a cyclotomic convolution. Correspondingly, the matrices A, B and C of the bilinear form also have a block diagonal structure.

The Split Nesting Algorithm

We now describe the split-nesting algorithm for general length circular convolution [link]. Let $n = p_1^{e_1} \cdots p_k^{e_k}$ where p_i are distinct primes. We have seen that

Equation:

$$S_n = P^t R^{-1} \left(\bigoplus_{d|n} \Psi\left(d\right) \right) RP$$

where P is the prime factor permutation $P = P_{p_1^{e_1},\ldots,p_k^{e_k}}$ and R represents the reduction operations. For example, see Equation 46 in Preliminaries. RP block diagonalizes S_n and each diagonal block represents a multi-dimensional cyclotomic convolution. To obtain a bilinear form for a multi-dimensional convolution, we can combine bilinear forms for one-dimensional convolutions. If $\left(A_{p_j^i}, B_{p_j^i}, C_{p_j^i}\right)$ describes a bilinear form for $\Phi_{p_j^i}\left(s\right)$ convolution and if

Equation:

$$\begin{aligned} A &= \oplus_{d|n} A_d \\ B &= \oplus_{d|n} B_d \\ C &= \oplus_{d|n} C_d \end{aligned}$$

with

Equation:

$$\begin{aligned} A_d &= \otimes_{p|d, p\in\mathscr{P}} A_{H_d(p)} \\ B_d &= \otimes_{p|d, p\in\mathscr{P}} B_{H_d(p)} \\ C_d &= \otimes_{p|d, p\in\mathscr{P}} C_{H_d(p)} \end{aligned}$$

where $H_d\left(p\right)$ is the highest power of p dividing d, and $\mathscr{P}$ is the set of primes, then $\left(ARP, BRP, P^t R^{-1} C\right)$ describes a bilinear form for n point

circular convolution. That is
Equation:

$$y = P^t R^{-1} C\{BRPh * ARPx\}$$

computes the circular convolution of h and x.

As above $\left(A_{p_j^i}, B_{p_j^i}, C_{p_j^i}\right)$ can be taken to be $\left(D_{\varphi\left(p_j^i\right)}, E_{\varphi\left(p_j^i\right)}, G_{p_j^i} F_{\varphi\left(p_j^i\right)}\right)$ where (D_d, E_d, F_d) describes a bilinear form for d point **linear** convolution. This is one particular choice for $\left(A_{p_j^i}, B_{p_j^i}, C_{p_j^i}\right)$ - other bilinear forms for cyclotomic convolution that are not derived from linear convolution algorithms exist.

Example:
A 45 point circular convolution algorithm:
Equation:

$$y = P^t R^{-1} C\{BRPh * ARPx\}$$

where
Equation:

$$\begin{aligned} P &= P_{9,5} \\ R &= R_{9,5} \\ A &= 1 \oplus A_3 \oplus A_9 \oplus A_5 \oplus (A_3 \otimes A_5) \oplus (A_9 \otimes A_5) \\ B &= 1 \oplus B_3 \oplus B_9 \oplus B_5 \oplus (B_3 \otimes B_5) \oplus (B_9 \otimes B_5) \\ C &= 1 \oplus C_3 \oplus C_9 \oplus C_5 \oplus (C_3 \otimes C_5) \oplus (C_9 \otimes C_5) \end{aligned}$$

and where $\left(A_{p_j^i}, B_{p_j^i}, C_{p_j^i}\right)$ describes a bilinear form for $\Phi_{p_j^i}(s)$ convolution.

The Matrix Exchange Property

The matrix exchange property is a useful technique that, under certain circumstances, allows one to save computation in carrying out the action of bilinear forms [link]. Suppose

Equation:

$$y = C\{Ax^*Bh\}$$

as in [link]. When h is known and fixed, Bh can be pre-computed so that y can be found using only the operations represented by C and A and the point by point multiplications denoted by *. The operation of B is absorbed into the multiplicative constants. Note that in [link], the matrix corresponding to C is more complicated than is B. It is therefore advantageous to absorb the work of C instead of B into the multiplicative constants if possible. This can be done when y is the circular convolution of x and h by using the matrix exchange property.

To explain the matrix exchange property we draw from [link]. Note that $y = Cdiag(Ax)Bh$, so that $\mathrm{Cdiag}(Ax)B$ must be the corresponding circulant matrix,

Equation:

$$\mathrm{Cdiag}\,(Ax)B = \begin{bmatrix} x_0 & x_{n-1} & \cdots & x_1 \\ x_1 & x_0 & & x_2 \\ \vdots & & & \\ x_{n-1} & x_{n-2} & & x_0 \end{bmatrix}.$$

Since $\mathrm{Cdiag}\,(Ax)B = J(\mathrm{Cdiag}(Ax)B)^tJ$ where J is the reversal matrix, one gets

Equation:

$$\begin{aligned} y &= C\{Ax^{*}Bh\} \\ &= \mathrm{C}\mathrm{diag}(Ax)Bh \\ &= J(\mathrm{C}\mathrm{diag}(Ax)B)^{t}Jh \\ &= JB^{t}diag\,(Ax)C^{t}Jh \\ &= JB^{t}\left\{Ax^{*}C^{t}Jh\right\} \end{aligned}$$

As noted in [link], the matrix exchange property can be used whenever $y = T(x)h$ where $T(x)$ satisfies $T\,(x) = J_1T(x)^{t}J_2$ for some matrices J_1 and J_2. In that case one gets $y = J_1B^{t}\left\{Ax^{*}C^{t}J_2h\right\}$.

Applying the matrix exchange property to [link] one gets
Equation:

$$y = JP^{t}R^{t}B^{t}\left\{C^{t}R^{-t}PJh^{*}ARPx\right\}.$$

Example:
A 45 point circular convolution algorithm:
Equation:

$$y = JP^{t}R^{t}B^{t}\{u^{*}ARPx\}$$

where $u = C^{t}R^{-t}PJh$ and
Equation:

$$\begin{aligned} P &= P_{9,5} \\ R &= R_{9,5} \\ A &= 1 \oplus A_3 \oplus A_9 \oplus A_5 \oplus (A_3 \otimes A_5) \oplus (A_9 \otimes A_5) \\ B^{t} &= 1 \oplus B_3^{t} \oplus B_9^{t} \oplus B_5^{t} \oplus \left(B_3^{t} \otimes B_5^{t}\right) \oplus \left(B_9^{t} \otimes B_5^{t}\right) \\ C^{t} &= 1 \oplus C_3^{t} \oplus C_9^{t} \oplus C_5^{t} \oplus \left(C_3^{t} \otimes C_5^{t}\right) \oplus \left(C_9^{t} \otimes C_5^{t}\right) \end{aligned}$$

and where $\left(A_{p_j^i}, B_{p_j^i}, C_{p_j^i}\right)$ describes a bilinear form for $\Phi_{p_j^i}(s)$ convolution.

A Bilinear Form for the DFT
This collection of modules is from a Rice University, ECE Department Technical Report written around September 1994. It grew out of the doctoral and post doctoral research of Ivan Selesnick working with Prof. C. Sidney Burrus at Rice. Earlier reports on this work were published in the ICASSP and ISCAS conference proceedings in 1992-94 and a fairly complete report was published in the IEEE Transaction on Signal Processing in January 1996.

A Bilinear Form for the DFT

A bilinear form for a prime length DFT can be obtained by making minor changes to a bilinear form for circular convolution. This relies on Rader's observation that a prime p point DFT can be computed by computing a $p-1$ point circular convolution and by performing some extra additions [link]. It turns out that when the Winograd or the split nesting convolution algorithm is used, only two extra additions are required. After briefly reviewing Rader's conversion of a prime length DFT in to a circular convolution, we will discuss a bilinear form for the DFT.

Rader's Permutation

To explain Rader's conversion of a prime p point DFT into a $p-1$ point circular convolution [link] we recall the definition of the DFT
Equation:

$$y(k) = \sum_{n=0}^{p-1} x(n) W^{kn}$$

with $W = \exp -\mathrm{j}\, 2\pi/p$. Also recall that a primitive root of p is an integer r such that $\langle r^m \rangle_p$ maps the integers $m = 0, \cdots, p-2$ to the integers $1, \cdots, p-1$. Letting $n = r^{-m}$ and $k = r^l$, where r^{-m} is the inverse of r^m modulo p, the DFT becomes
Equation:

$$y\left(r^{l}\right)=x\left(0\right)+\sum_{m=0}^{p-2}x\left(r^{-m}\right)W^{r^{l}r^{-m}}$$

for $l=0,\cdots,p-2$. The `DC' term fis given by $y\left(0\right)=\sum_{n=0}^{p-1}x\left(n\right)$. By defining new functions
Equation:

$$x'\left(m\right)=x\left(r^{-m}\right),\quad y'\left(m\right)=y\left(r^{m}\right),\quad W'\left(m\right)=W^{r^{m}}$$

which are simply permuted versions of the original sequences, the DFT becomes
Equation:

$$y'\left(l\right)=x\left(0\right)+\sum_{m=0}^{p-2}x'\left(m\right)W'\left(l-m\right)$$

for $l=0,\cdots,p-2$. This equation describes circular convolution and therefore any circular convolution algorithm can be used to compute a prime length DFT. It is only necessary to (i) permute the input, the roots of unity and the output, (ii) add $x(0)$ to each term in [link] and (iii) compute the DC term.

To describe a bilinear form for the DFT we first define a permutation matrix Q for the permutation above. If p is a prime and r is a primitive root of p, then let Q_r be the permutation matrix defined by
Equation:

$$Qe_{\left\langle r^{k}\right\rangle _{p}-1}=e_{k}$$

for $0\leq k\leq p-2$ where e_k is the k^{th} standard basis vector. Let the $\widetilde{w}$ be a $p-1$ point vector of the roots of unity:
Equation:

$$\widetilde{w} = \left(W^1, \cdots, W^{p-1}\right)^t.$$

If s is the inverse of r modulo p (that is, $rs = 1$ modulo p) and $\widetilde{x} = (x(1), \cdots, x(p-1))^t$, then the circular convolution of [link] can be computed with the bilinear form of [link]:
Equation:

$$Q_s^t J P^t R^t B^t \left\{C^t R^{-t} P J Q_s \widetilde{w} * A R P Q_r \widetilde{x}\right\}.$$

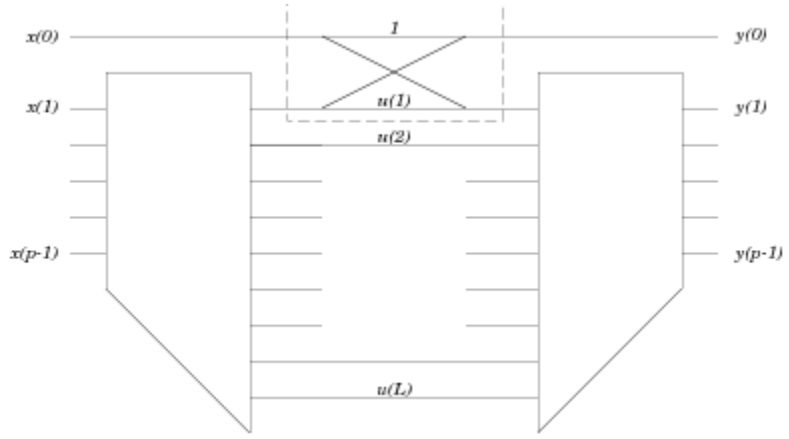
This bilinear form does not compute $y(0)$, the DC term. Furthermore, it is still necessary to add the $x(0)$ term to each of the elements of [link] to obtain $y(1), \cdots, y(p-1)$.

Calculation of the DC term

The computation of $y(0)$ turns out to be very simple when the bilinear form [link] is used to compute the circular convolution in [link]. The first element of $ARPQ_r\widetilde{x}$ in [link] is the residue modulo the polynomial $s - 1$, that is, the first element of this vector is the sum of the elements of $\widetilde{x}$. (The first row of the matrix, R, representing the reduction operation is a row of 1's, and the matrices P and Q_r are permutation matrices.) Therefore, the DC term can be computed by adding the first element of $ARPQ_r\widetilde{x}$ to $x(0)$. Hence, when the Winograd or split nesting algorithm is used to perform the circular convolution of [link], the computation of the DC term requires only one extra complex addition for complex data.

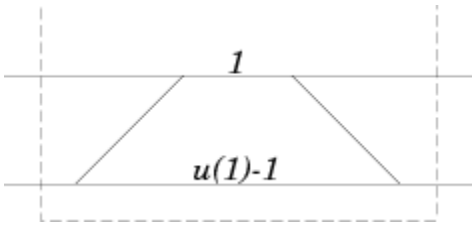
The addition $x(0)$ to each of the elements of [link] also requires only one complex addition. By adding $x(0)$ to the first element of $\left\{C^t R^{-t} P J Q_s \widetilde{w} * A R P Q_r \widetilde{x}\right\}$ in [link] and applying $Q_s^t J P^t R^t$ to the result, $x(0)$ is added to each element. (Again, this is because the first column of R^t is a column of 1's, and the matrices Q_s^t, J and P^t are permutation matrices.)

Although the DFT can be computed by making these two extra additions, this organization of additions does not yield a bilinear form. However, by making a minor modification, a bilinear form can be retrieved. The method described above can be illustrated in [link] with $u = C^t R^{-t} P J Q_s \widetilde{w}$.



The flow graph for the computation of the DFT.

Clearly, the structure highlighted in the dashed box can be replaced by the structure in [link].



The flow graph for the bilinear form.

By substituting the second structure for the first, a bilinear form is obtained. The resulting bilinear form for a prime length DFT is
Equation:

$$y = \begin{bmatrix} 1 & \\ & Q_s^t J P^t R^t B^t \end{bmatrix} U_p^t \left\{ V_p \begin{bmatrix} 1 & \\ & C^t R^{-t} P J Q_s \end{bmatrix} w * U_p \begin{bmatrix} 1 & \\ & A R P Q_r \end{bmatrix} x \right\}$$

where $w = \left(W^0, \cdots, W^{p-1}\right)^t$, $x = \left(x\left(0\right), \cdots, x\left(p-1\right)\right)^t$, and where U_p is the matrix with the form
Equation:

$$U_p = \begin{matrix} 1 & 1 & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & & & 1 \end{matrix}$$

and V_p is the matrix with the form
Equation:

$$U_p = \begin{matrix} 1 & & & & & \\ -1 & 1 & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & & & 1 \end{matrix}$$

Implementing Kronecker Products Efficiently
This collection of modules is from a Rice University, ECE Department Technical Report written around September 1994. It grew out of the doctoral and post doctoral research of Ivan Selesnick working with Prof. C. Sidney Burrus at Rice. Earlier reports on this work were published in the ICASSP and ISCAS conference proceedings in 1992-94 and a fairly complete report was published in the IEEE Transaction on Signal Processing in January 1996.

Implementing Kronecker Products Efficiently

In the algorithm described above we encountered expressions of the form $A_1 \otimes A_2 \otimes \cdots \otimes A_n$ which we denote by $\otimes_{i=1}^{n} A_i$. To calculate the product $(\otimes_i A_i)x$ it is computationally advantageous to factor $\otimes_i A_i$ into terms of the form $I \otimes A_i \otimes I$[link]. Then each term represents a set of copies of A_i. First, recall the following property of Kronecker products
Equation:

$$AB \otimes CD = (A \otimes C)(B \otimes D).$$

This property can be used to factor $\otimes_i A_i$ in the following way. Let the number of rows and columns of A_i be denoted by r_i and c_i respectively. Then
Equation:

$$\begin{aligned} A_1 \otimes A_2 &= A_1 I_{c_1} \otimes I_{r_2} A_2 \\ &= (A_1 \otimes I_{r_2})\,(I_{c_1} \otimes A_2). \end{aligned}$$

But we can also write
Equation:

$$\begin{aligned} A_1 \otimes A_2 &= I_{r_1} A_1 \otimes A_2 I_{c_2} \\ &= (I_{r_1} \otimes A_2)\,(A_1 \otimes I_{c_2}). \end{aligned}$$

Note that in factorization [link], copies of A_2 are applied to the data vector x first, followed by copies of A_1. On the other hand, in factorization [link], copies of A_1 are applied to the data vector x first, followed by copies of A_2 . These two factorizations can be distinguished by the sequence in which A_1 and A_2 are ordered.

Lets compare the computational complexity of factorizations [link] and [link]. Notice that [link] consists of r_2 copies of A_1 and c_1 copies of A_2, therefore [link] has a computational cost of $r_2\mathscr{Q}_1 + c_1\mathscr{Q}_2$ where $\mathscr{Q}_i$ is the computational cost of A_i. On the other hand, the computational cost of [link] is $c_2\mathscr{Q}_1 + r_1\mathscr{Q}_2$. That is, the factorizations [link] and [link] have in general **different** computational costs when A_i are not square. Further, observe that [link] is the more efficient factorization exactly when
Equation:

$$r_2\mathscr{Q}_1 + c_1\mathscr{Q}_2 < c_2\mathscr{Q}_1 + r_1\mathscr{Q}_2$$

or equivalently, when
Equation:

$$\frac{r_1 - c_1}{\mathscr{Q}_1} > \frac{r_2 - c_2}{\mathscr{Q}_2}.$$

Consequently, in the more efficient factorization, the operation A_i applied to the data vector x first is the one for which the ratio $(r_i - c_i)/\mathscr{Q}_i$ is the more negative. If $r_1 > c_1$ and $r_2 < c_2$ then [link] is always true ($\mathscr{Q}_i$ is always positive). Therefore, in the most computationally efficient factorization of $A_1 \otimes A_2$, matrices with fewer rows than columns are always applied to the data vector x before matrices with more rows than columns. If both matrices are square, then their ordering does not affect the computational efficiency, because in that case each ordering has the same computation cost.

We now consider the Kronecker product of more than two matrices. For the Kronecker product $\otimes_{i=1}^{n} A_i$ there are $n!$ possible different ways in which to order the operations A_i. For example

Equation:

$$\begin{aligned} A_1 \otimes A_2 \otimes A_3 &= (A_1 \otimes I_{r_2 r_3})\,(I_{c_1} \otimes A_2 \otimes I_{r_3})\,(I_{c_1 c_2} \otimes A_3) \\ &= (A_1 \otimes I_{r_2 r_3})\,(I_{c_1 r_2} \otimes A_3)\,(I_{c_1} \otimes A_2 \otimes I_{c_3}) \\ &= (I_{r_1} \otimes A_2 \otimes I_{r_3})\,(A_1 \otimes I_{c_2 r_3})\,(I_{c_1 c_2} \otimes A_3) \\ &= (I_{r_1} \otimes A_2 \otimes I_{r_3})\,(I_{r_1 c_2} \otimes A_3)\,(A_1 \otimes I_{c_2 c_3}) \\ &= (I_{r_1 r_2} \otimes A_3)\,(A_1 \otimes I_{r_2 c_3})\,(I_{c_1} \otimes A_2 \otimes I_{c_3}) \\ &= (I_{r_1 r_2} \otimes A_3)\,(I_{r_1} \otimes A_2 \otimes I_{c_3})\,(A_1 \otimes I_{c_2 c_3}) \end{aligned}$$

Each factorization of $\otimes_i A_i$ can be described by a permutation $g(\cdot)$ of $\{1, \ldots, n\}$ which gives the order in which A_i is applied to the data vector x. $A_{g(1)}$ is the first operation applied to the data vector x, $A_{g(2)}$ is the second, and so on. For example, the factorization [link] is described by the permutation $g(1) = 3$, $g(2) = 1$, $g(3) = 2$. For $n = 3$, the computational cost of each factorization can be written as
Equation:

$$\mathscr{C}(g) = \mathscr{Q}_{g(1)} c_{g(2)} c_{g(3)} + r_{g(1)} \mathscr{Q}_{g(2)} c_{g(3)} + r_{g(1)} r_{g(2)} \mathscr{Q}_{g(3)}$$

In general
Equation:

$$\mathscr{C}(g) = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} r_{g(j)} \right) \mathscr{Q}_{g(i)} \left(\prod_{j=i+1}^{n} c_{g(j)} \right).$$

Therefore, the most efficient factorization of $\otimes_i A_i$ is described by the permutation $g(\cdot)$ that minimizes $\mathscr{C}$.

It turns out that for the Kronecker product of more than two matrices, the ordering of operations that describes the most efficient factorization of $\otimes_i A_i$ also depends only on the ratios $(r_i - c_i)/\mathscr{Q}_i$. To show that this is the case, suppose $u(\cdot)$ is the permutation that minimizes $\mathscr{C}$, then $u(\cdot)$ has the property that

Equation:

$$\frac{r_{u(k)} - c_{u(k)}}{\mathscr{Q}_{u(k)}} \leq \frac{r_{u(k+1)} - c_{u(k+1)}}{\mathscr{Q}_{u(k+1)}}$$

for $k = 1, \cdots, n-1$. To support this, note that since $u(\cdot)$ is the permutation that minimizes $\mathscr{C}$, we have in particular
Equation:

$$\mathscr{C}(u) \leq \mathscr{C}(v)$$

where $v(\cdot)$ is the permutation defined by the following:
Equation:

$$v\left(i\right) = \begin{cases} u(i) & i < k, i > k+1 \\ u(k+1) & i = k \\ u(k) & i = k+1 \end{cases}.$$

Because only two terms in [link] are different, we have from [link]
Equation:

$$\sum_{i=k}^{k+1} \left(\prod_{j=1}^{i-1} r_{u(j)} \right) \mathscr{Q}_{u(i)} \left(\prod_{j=i+1}^{n} c_{u(j)} \right) \leq \sum_{i=k}^{k+1} \left(\prod_{j=1}^{i-1} r_{v(j)} \right) \mathscr{Q}_{v(i)} \left(\prod_{j=i+1}^{n} c_{v(j)} \right)$$

which, after canceling common terms from each side, gives
Equation:

$$\mathscr{Q}_{u(k)} c_{u(k+1)} + r_{u(k)} \mathscr{Q}_{u(k+1)} \leq \mathscr{Q}_{v(k)} c_{v(k+1)} + r_{v(k)} \mathscr{Q}_{v(k+1)}.$$

Since $v(k) = u(k+1)$ and $v(k+1) = u(k)$ this becomes
Equation:

$$\mathcal{Q}_{u(k)}c_{u(k+1)} + r_{u(k)}\mathcal{Q}_{u(k+1)} \leq \mathcal{Q}_{u(k+1)}c_{u(k)} + r_{u(k+1)}\mathcal{Q}_{u(k)}$$

which is equivalent to [link]. Therefore, to find the best factorization of $\otimes_i A_i$ it is necessary only to compute the ratios $(r_i - c_i)/\mathcal{Q}_i$ and to order them in an non-decreasing order. The operation A_i whose index appears first in this list is applied to the data vector x first, and so on

As above, if $r_{u(k+1)} > c_{u(k+1)}$ and $r_{u(k)} < c_{u(k)}$ then [link] is always true. Therefore, in the most computationally efficient factorization of $\otimes_i A_i$, all matrices with fewer rows than columns are always applied to the data vector x before any matrices with more rows than columns. If some matrices are square, then their ordering does not affect the computational efficiency as long as they are applied **after** all matrices with fewer rows than columns and **before** all matrices with more rows than columns.

Once the permutation $g(\cdot)$ that minimizes $\mathscr{C}$ is determined by ordering the ratios $(r_i - c_i)/\mathcal{Q}_i$, $\otimes_i A_i$ can be written as
Equation:

$$\bigotimes_{i=1}^{n} A_i = \prod_{i=n}^{1} I_{a(i)} \otimes A_{g(i)} \otimes I_{b(i)}$$

where
Equation:

$$a\left(i\right) = \prod_{k=1}^{g(i)-1} \gamma\left(i,k\right)$$

Equation:

$$b\left(i\right) = \prod_{k=g(i)+1}^{n} \gamma\left(i,k\right)$$

and where $\gamma(\cdot)$ is defined by
Equation:

$$\gamma\left(i,k\right)=\begin{cases}r_k & \text{if } g(i)>g(k)\\ c_k & \text{if } g(i)<g(k)\end{cases}.$$

Some Matlab Code

A Matlab program that computes the permutation that describes the computationally most efficient factorization of $\otimes_{i=1}^{n}A_i$ is `cgc()` . It also gives the resulting computational cost. It requires the computational cost of each of the matrices A_i and the number of rows and columns of each.

```
function [g,C] = cgc(Q,r,c,n)
% [g,C] = cgc(Q,r,c,n);
% Compute g and C
% g : permutation that minimizes C
% C : computational cost of Kronecker product of
A(1),...,A(n)
% Q : computation cost of A(i)
% r : rows of A(i)
% c : columns of A(i)
% n : number of terms
f = find(Q==0);
Q(f) = eps * ones(size(Q(f)));
Q = Q(:);
r = r(:);
c = c(:);
[s,g] = sort((r-c)./Q);
C = 0;
for i = 1:n
   C = C + prod(r(g(1:i-
1)))*Q(g(i))*prod(c(g(i+1:n)));
end
C = round(C);
```

The Matlab program `kpi()` implements the Kronecker product $\otimes_{i=1}^{n} A_i$.

```
function y = kpi(d,g,r,c,n,x)
% y = kpi(d,g,r,c,n,x);
% Kronecker Product : A(d(1)) kron ... kron
A(d(n))
% g : permutation of 1,...,n
% r : [r(1),...,r(n)]
% c : [c(1),..,c(n)]
% r(i) : rows of A(d(i))
% c(i) : columns of A(d(i))
% n : number of terms
for i = 1:n
   a = 1;
   for k = 1:(g(i)-1)
      if i > find(g==k)
         a = a * r(k);
      else
         a = a * c(k);
      end
   end
   b = 1;
   for k = (g(i)+1):n
      if i > find(g==k)
         b = b * r(k);
      else
         b = b * c(k);
      end
   end
   % y = (I(a) kron A(d(g(i))) kron I(b)) * x;
   y = IAI(d(g(i)),a,b,x);
end
```

where the last line of code calls a function that implements $\left(I_a \otimes A_{d(g(i))} \otimes I_b\right)x$. That is, the program `IAI(i,a,b,x)` implements $\left(I_a \otimes A\left(i\right) \otimes I_b\right)x$.

The Matlab program `IAI` implements $y = (I_m \otimes A \otimes I_n)x$

```
function y = IAI(A,r,c,m,n,x)
% y = (I(m) kron A kron I(n))x
% r : number of rows of A
% c : number of columns of A
v = 0:n:n*(r-1);
u = 0:n:n*(c-1);
for i = 0:m-1
   for j = 0:n-1
      y(v+i*r*n+j+1) = A * x(u+i*c*n+j+1);
   end
end
```

It simply uses two loops to implement the mn copies of A. Each copy of A is applied to a different subset of the elements of x.

Vector/Parallel Interpretation

The command $I \otimes A \otimes I$ where $\otimes$ is the Kronecker (or Tensor) product can be interpreted as a vector/parallel command [link], [link]. In these references, the implementation of these commands is discussed in detail and they have found that the Tensor product is "an extremely useful tool for matching algorithms to computer architectures [link]."

The expression $I \otimes A$ can easily be seen to represent a parallel command:
Equation:

$$I \otimes A = \begin{bmatrix} A & & \\ & A & \\ & & \ddots \\ & & & A \end{bmatrix}.$$

Each block along the diagonal acts on non-overlapping sections of the data vector - so that each section can be performed in parallel. Since each section represents exactly the same operation, this form is amenable to implementation on a computer with a parallel architectural configuration. The expression $A \otimes I$ can be similarly seen to represent a vector command, see [link].

It should also be noted that by employing `stride' permutations, the command $(I \otimes A \otimes I)x$ can be replaced by either $(I \otimes A)x$ or $(A \otimes I)x$ [link], [link]. It is only necessary to permute the input and output. It is also the case that these stride permutations are natural loading and storing commands for some architectures.

In the programs we have written in conjunction with this paper we implement the commands $y = (I \otimes A \otimes I)x$ with loops in a set of subroutines. The circular convolution and prime length FFT programs we present, however, explicitly use the form $I \otimes A \otimes I$ to make clear the structure of the algorithm, to make them more modular and simpler, and to make them amenable to implementation on special architectures. In fact, in [link] it is suggested that it might be practical to develop tensor product compilers. The FFT programs we have generated will be well suited for such compilers.

Programs for Circular Convolution

To write a program that computes the circular convolution of h and x using the bilinear form Equation 24 in Bilinear Forms for Circular Convolution we need subprograms that carry out the action of P, P^t, R, R^t, A and B^t. We are assuming, as is usually done, that h is fixed and known so that $u = C^t R^{-t} P J h$ can be pre-computed and stored. To compute these multiplicative constants u we need additional subprograms to carry out the action of C^t and R^{-t} but the efficiency with which we compute u is unimportant since this is done beforehand and u is stored.

In Prime Factor Permutations we discussed the permutation P and a program for it `pfp()` appears in the appendix. The reduction operations R, R^t and R^{-t} we have described in Reduction Operations and programs for these reduction operations `KRED()` etc, also appear in the appendix. To carry out the operation of A and B^t we need to be able to carry out the action of $A_{d_1} \otimes \cdots \otimes A_{d_k}$ and this was discussed in Implementing Kronecker Products Efficiently. Note that since A and B^t are block diagonal, each diagonal block can be done separately. However, since they are rectangular, it is necessary to be careful so that the correct indexing is used.

To facilitate the discussion of the programs we generate, it is useful to consider an example. Take as an example the 45 point circular convolution algorithm listed in the appendix. From Equation 19 from Bilinear Forms for Circular Convolution we find that we need to compute $x = P_{9,5}x$ and $x = R_{9,5}x$. These are the first two commands in the program.

We noted above that bilinear forms for linear convolution, (D_d, E_d, F_d), can be used for these cyclotomic convolutions. Specifically we can take $A_{p^i} = D_{\varphi(p^i)}$, $B_{p^i} = E_{\varphi(p^i)}$ and $C_{p^i} = G_{p^i} F_{\varphi(p^i)}$. In this case Equation 20 in Bilinear Forms for Circular Convolution becomes
Equation:

$$A = 1 \oplus D_2 \oplus D_6 \oplus D_4 \oplus (D_2 \otimes D_4) \oplus (D_6 \otimes D_4).$$

In our approach this is what we have done. When we use the bilinear forms for convolution given in the appendix, for which $D_4 = D_2 \otimes D_2$ and $D_6 = D_2 \otimes D_3$, we get
Equation:

$$A = 1 \oplus D_2 \oplus (D_2 \otimes D_3) \oplus (D_2 \otimes D_2) \oplus (D_2 \otimes D_2 \otimes D_2) \oplus (D_2 \otimes D_3 \otimes D_2 \otimes D_2)$$

and since $E_d = D_d$ for the linear convolution algorithms listed in the appendix, we get
Equation:

$$B = 1 \oplus D_2^t \oplus \left(D_2^t \otimes D_3^t\right) \oplus \left(D_2^t \otimes D_2^t\right) \oplus \left(D_2^t \otimes D_2^t \otimes D_2^t\right) \oplus \left(D_2^t \otimes D_3^t \otimes D_2^t \otimes D_2^t\right).$$

From the discussion above, we found that the Kronecker products like $D_2 \otimes D_2 \otimes D_2$ appearing in these expressions are best carried out by factoring the product in to factors of the form $I_a \otimes D_2 \otimes I_b$. Therefore we need a program to carry out $(I_a \otimes D_2 \otimes I_b)x$ and $(I_a \otimes D_3 \otimes I_b)x$. These function are called `ID2I(a,b,x)` and `ID3I(a,b,x)` and are listed in the appendix. The transposed form, $\left(I_a \otimes D_2^t \otimes I_b\right)x$, is called `ID2tI(a,b,x)`.

To compute the multiplicative constants we need C^t. Using $C_{p^i} = G_{p^i} F_{\varphi(p^i)}$ we get
Equation:

$$\begin{aligned} C^t &= 1 \oplus F_2^t G_3^t \oplus F_6^t G_9^t \oplus F_4^t G_5^t \oplus \left(F_2^t G_3^t \otimes F_4^t G_5^t\right) \oplus \left(F_6^t G_9^t \otimes F_4^t G_5^t\right) \\ &= 1 \oplus F_2^t G_3^t \oplus F_6^t G_9^t \oplus F_4^t G_5^t \oplus \left(F_2^t \otimes F_4^t\right)\left(G_3^t \otimes G_5^t\right) \oplus \left(F_6^t \otimes F_4^t\right)\left(G_9^t \otimes G_5^t\right). \end{aligned}$$

The Matlab function `KFt` carries out the operation $F_{d_1} \otimes \cdots F_{d_K}$. The Matlab function `Kcrot` implements the operation $G_{p_1^{e_1}} \otimes \cdots G_{p_K^{e_K}}$. They are both listed in the appendix.

Common Functions

By recognizing that the convolution algorithms for different lengths share a lot of the same computations, it is possible to write a set of programs that take advantage of this. The programs we have generated call functions from a relatives small set. Each program calls these functions with different arguments, in differing orders, and a different number of times. By organizing the program structure in a modular way, we are able to generate relatively compact code for a wide variety of lengths.

In the appendix we have listed code for the following functions, from which we create circular convolution algorithms. In the next section we generate FFT programs using this same set of functions.

- **Prime Factor Permutations**The Matlab function `pfp` implements this permutation of Prime Factor Permutations. Its transpose is implemented by `pfpt` .
- **Reduction Operations**The Matlab function `KRED` implements the reduction operations of Reduction Operations. Its transpose is implemented by `tKRED` . Its inverse transpose is implemented by `itKRED` and this function is used only for computing the multiplicative constants.
- **Linear Convolution Operations**`ID2I` and `ID3I` are Matlab functions for the operations $I \otimes D_2 \otimes I$ and $I \otimes D_3 \otimes I$. These linear convolution operations are also described in the appendix `Bilinear Forms for Linear Convolution.' `ID2tI` and `ID3tI` implement the transposes, $I \otimes D_2^t \otimes I$ and $I \otimes D_3^t \otimes I$.

Operation Counts

[link] lists operation counts for some of the circular convolution algorithms we have generated. The operation counts do not include any arithmetic operations involved in the index variable or loops. They include only the arithmetic operations that involve the data sequence x in the convolution of x and h.

The table in [link] for the split nesting algorithm gives very similar arithmetic operation counts. For all lengths not divisible by 9, the algorithms we have developed use the same number of multiplications and the same number or fewer additions. For lengths which are divisible by 9, the algorithms described in [link] require fewer additions than do ours. This is because the algorithms whose operation counts are tabulated in the table in [link] use a special $\Phi_9(s)$ convolution algorithm. It should be noted, however, that the efficient $\Phi_9(s)$ convolution algorithm of [link] is not constructed from smaller algorithms using the Kronecker product, as is ours. As we have discussed above, the use of the Kronecker product facilitates adaptation to special computer architectures and yields a very compact program with function calls to a small set of functions.

N	muls	adds		N	muls	adds		N	muls	adds		N	m
2	2	4		24	56	244		80	410	1546		240	1(
3	4	11		27	94	485		84	320	1712		252	1!
4	5	15		28	80	416		90	380	1858		270	1!
5	10	31		30	80	386		105	640	2881		280	2!

6	8	34		35	160	707		108	470	2546		315	3(
7	16	71		36	95	493		112	656	2756		336	2(
8	14	46		40	140	568		120	560	2444		360	2(
9	19	82		42	128	718		126	608	3378		378	3(
10	20	82		45	190	839		135	940	4267		420	3:
12	20	92		48	164	656		140	800	3728		432	3(
14	32	170		54	188	1078		144	779	3277		504	4:
15	40	163		56	224	1052		168	896	4276		540	4'
16	41	135		60	200	952		180	950	4466		560	6(
18	38	200		63	304	1563		189	1504	7841		630	6(
20	50	214		70	320	1554		210	1280	6182		720	7'
21	64	317		72	266	1250		216	1316	6328		756	7(

Operation counts for split nesting circular convolution algorithms

It is possible to make further improvements to the operation counts given in [link][link], [link]. Specifically, algorithms for prime power cyclotomic convolution based on the polynomial transform, although more complicated, will give improvements for the longer lengths listed [link], [link]. These improvements can be easily included in the code generating program we have developed.

Programs for Prime Length FFTs

Using the circular convolution algorithms described above, we can easily design algorithms for prime length FFTs. The only modifications that needs to be made involve the permutation of Rader [link] and the correct calculation of the DC term ($y(0)$). These modifications are easily made to the above described approach. It simply requires a few extra commands in the programs. Note that the multiplicative constants are computed directly, since we have programs for all the relevant operations.

In the version we have currently implemented and verified for correctness, we precompute the multiplicative constants, the input permutation and the output permutation. From Equation 8 from A Bilinear Form for the DFT, the multiplicative constants are given by $V_p \left(1 \oplus C^t R^{-t} P J Q_s\right) w$, the input permutation is given by $1 \oplus P Q_r$, and the output permutation is given by $1 \oplus Q_s^t J P^t$. The multiplicative constants, the input and output permutation are each stored as vectors. These vectors are then passed to the prime length FFT program, which consists of the appropriate function calls, see the appendix. In previous prime length FFT modules, the input and output permutations can be completely absorbed in to the computational instructions. This is possible because they are written as straight line code. It is simple to modify the code generating program we have implemented so that it produces straight line code and absorbs the permutations in to the computational program instructions.

In an in-place in-order prime factor algorithm for the DFT [link], [link], the necessary permuted forms of the DFT can be obtained by modifying the multiplicative constants. This can be easily done by permuting the roots of unity, w, in the expression for the multiplicative constants [link], [link], nothing else in the structure of the algorithm needs to be changed. By changing the multiplicative constants, it is not possible, however, to omit the permutation required for Rader's conversion of the prime length DFT in to circular convolution.

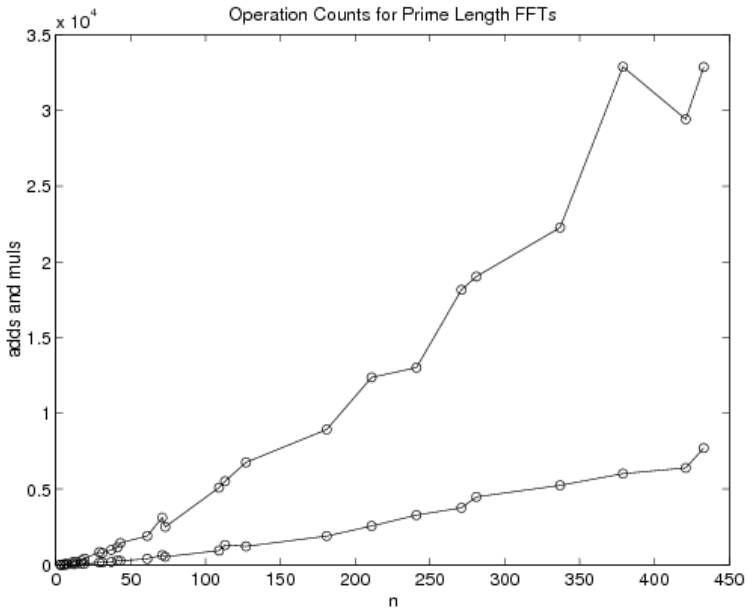
Operation Counts

[link] lists the arithmetic operations incurred by the FFT programs we have generated and verified for correctness. Note that the number of additions and multiplications incurred by the programs we have generated are the same as previously existing programs for prime lengths up to and including 13. For $p = 17$ a program with 70 multiplications and 314 additions has been written, and for $p = 19$ a program with 76 multiplications and 372 additions has been written [link]. Thus for the length $p = 17$, the program we have generated requires fewer total arithmetic operations, while for $p = 19$, ours uses more.

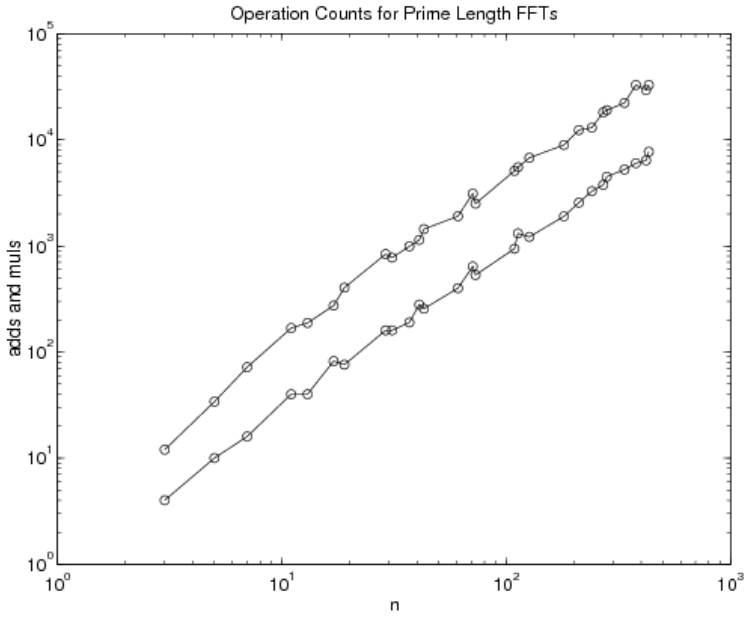
There are several table of operation counts in [link], each table corresponding to a different variation of the algorithms used in that paper. For each variation, the algorithms we have described use fewer additions and fewer multiplications. The focus of [link], however, is the implementation of prime point FFT on various computer architectures and the advantage that can be gained from matching algorithms with architectures. It should be noted that the highest prime in [link] for which an FFT was designed is 29. Although we have not executed the programs described in this paper on these computers, they are, as mentioned above, written to be easily adapted to parallel/vector computers.

P	muls	adds		P	muls	adds		P	muls	adds
3	4	12		41	280	1140		241	3280	13020
5	10	34		43	256	1440		271	3760	18152
7	16	72		61	400	1908		281	4480	19036
11	40	168		71	640	3112		337	5248	22268
13	40	188		73	532	2504		379	6016	32880
17	82	274		109	940	5096		421	6400	29412
19	76	404		113	1312	5516		433	7708	32864
29	160	836		127	1216	6760		541	9400	43020
31	160	776		181	1900	8936		631	12160	56056
37	190	990		211	2560	12368		757	15040	76292

Operation counts for prime length FFTs



Plot of additions and multiplications incurred by prime length FFTs.



Plot of additions and multiplications incurred by prime length FFTs.

Conclusion

We have found that by using the split nesting algorithm for circular convolution a new set of efficient prime length DFT modules that cover a wide variety of lengths can be developed. We have also exploited the structure in the split nesting algorithm to write a program that automatically generates compact readable code for convolution and prime length FFT programs.

The resulting code makes clear the organization and structure of the algorithm and clearly enumerates the disjoint convolutions into which the problem is decomposed. These independent convolutions can be executed in parallel and, moreover, the individual commands are of the form $I \otimes A \otimes I$ which can be executed as parallel/vector commands on appropriate computer architectures[link]. By recognizing also that the algorithms for different lengths share many of the same computational structures, the code we generate is made up of calls to a relatively small set of functions. Accordingly, the subroutines can be designed to specifically suit a given architecture.

The number of additions and multiplications incurred by the programs we have generated are the same as or are competitive with existing prime length FFT programs. We note that previously, prime length FFTs were made available for primes only up to 29. As in the original Winograd short convolution algorithms, the efficiency of the resulting prime p point DFT algorithm depends largely upon the factorability of $p-1$. For example, if $p-1$ is two times a prime, then an efficient p point DFT algorithm is more difficult to develop.

It should be noted too that the programs for convolution developed above are useful in the convolution of long integer sequences when exact results are needed. This is because all multiplicative constants in an n point integer convolution are integer multiples of $1/n$ and this division by n can be delayed until the last stage or can simply be omitted if a scaled version of the convolution is acceptable.

By developing a large library of prime point FFT programs we can extend the maximum length and the variety of lengths of a prime factor algorithm or a Winograd Fourier transform algorithm. Furthermore, because the approach taken in this paper gives a bilinear form, it can be incorporated into the dynamic programming technique for designing optimal composite length FFT algorithms [link]. The programs described in this paper can also be adapted to obtain discrete cosine transform (DCT) algorithms by simply permuting the input and output sequences [link].

Appendix: Bilinear Forms for Linear Convolution

The following is a collection of bilinear forms for linear convolution. In each case (D_n, D_n, F_n) describes a bilinear form for n point linear convolution. That is
Equation:

$$y = F_n\{D_n h * D_n x\}$$

computes the linear convolution of the n point sequences h and x.

For each D_n we give Matlab programs that compute $D_n x$ and $D_n^t x$, and for each F_n we give a Matlab program that computes $F_n^t x$. When the matrix exchange algorithm is employed in the design of circular convolution algorithms, these are the relevant operations.

2 point linear convolution

D_2 can be implemented with 1 addition, D_2^t with two additions.
Equation:

$$D_2 = \begin{matrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{matrix}$$

Equation:

$$F_2 = \begin{matrix} 1 & 0 & 0 \\ -1 & -1 & 1 \\ 0 & 1 & 0 \end{matrix}$$

```
function y = D2(x)
y = zeros(3,1);
y(1) = x(1);
y(2) = x(2);
y(3) = x(1) + x(2);
```

```
function y = D2t(x)
y = zeros(2,1);
y(1) = x(1)+x(3);
y(2) = x(2)+x(3);
```

```
function y = F2t(x)
y = zeros(3,1);
y(1) = x(1)-x(2);
y(2) = -x(2)+x(3);
y(3) = x(2);
```

3 point linear convolution

D_3 can be implemented in 7 additions, D_3^t in 9 additions.
Equation:

$$D_3 = \begin{matrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{matrix}$$

Equation:

$$F_3 = \frac{1}{6} \begin{matrix} 6 & 0 & 0 & 0 & 0 \\ -3 & 6 & -2 & -1 & 12 \\ -6 & 3 & 3 & 0 & -6 \\ 3 & -3 & -1 & 1 & -12 \\ 0 & 0 & 0 & 0 & 6 \end{matrix}$$

```
function y = D3(x)
y = zeros(5,1);
a = x(2)+x(3);
b = x(3)-x(2);
y(1) = x(1);
y(2) = x(1)+a;
y(3) = x(1)+b;
y(4) = a+a+b+y(2);
y(5) = x(3);
```

```
function y = D3t(x)
y = zeros(3,1);
y(1) = x(2)+x(3)+x(4);
a = x(4)+x(4);
y(2) = x(2)-x(3)+a;
y(3) = y(1)+x(4)+a;
y(1) = y(1)+x(1);
y(3) = y(3)+x(5);
```

```
function y = F3t(x)
y = zeros(5,1);
y(1) = 6*x(1)-3*x(2)-6*x(3)+3*x(4);
y(2) = 6*x(2)+3*x(3)-3*x(4);
y(3) = -2*x(2)+3*x(3)-x(4);
y(4) = -x(2)+x(4);
y(5) = 12*x(2)-6*x(3)-12*x(4)+6*x(5);
y = y/6;
```

4 point linear convolution

Equation:

$$D_4 = D_2 \otimes D_2$$

Equation:

$$F_4 = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 \\ 0 & -1 & 0 & 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{matrix}$$

```
function y = F4t(x)
y = zeros(7,1);
y(1) = x(1)-x(2)-x(3)+x(4);
y(2) = -x(2)+x(3)+x(4)-x(5);
y(3) = x(2)-x(4);
y(4) = -x(3)+x(4)+x(5)-x(6);
y(5) = x(4)-x(5)-x(6)+x(7);
y(6) = -x(4)+x(6);
y(7) = x(3)-x(4);
y(8) = -x(4)+x(5);
y(9) = x(4);
```

6 point linear convolution

Equation:

$$D_6 = D_2 \otimes D_3$$

Equation:

$$F_6 = \frac{1}{6} \begin{matrix} 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 6 & -2 & -1 & 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -6 & 3 & 3 & 0 & -6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & -3 & -1 & 1 & -12 & -6 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 3 & -6 & 2 & 1 & -6 & 3 & -6 & 2 & 1 & -12 & -3 & 6 & -2 & -1 & 12 \\ 6 & -3 & -3 & 0 & 6 & 6 & -3 & -3 & 0 & 6 & -6 & 3 & 3 & 0 & -6 \\ -3 & 3 & 1 & -1 & 12 & 3 & 3 & 1 & -1 & 12 & 3 & -3 & -1 & 1 & -12 \\ 0 & 0 & 0 & 0 & -6 & -3 & 6 & -2 & -1 & 6 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & -6 & 3 & 3 & 0 & -6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & -3 & -1 & 1 & -12 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

```
function y = F6t(x)
y = zeros(15,1);
y(1) = 6*x(1)-3*x(2)-6*x(3)-3*x(4)+3*x(5)+6*x(6)-3*x(7);
y(2) = 6*x(2)+3*x(3)-3*x(4)-6*x(5)-3*x(6)+3*x(7);
y(3) = -2*x(2)+3*x(3)-x(4)+2*x(5)-3*x(6)+x(7);
y(4) = -x(2)+x(4)+x(5)-x(7);
y(5) = 12*x(2)-6*x(3)-12*x(4)-6*x(5)+6*x(6)+12*x(7)-6*x(8);
y(6) = -6*x(4)+3*x(5)+6*x(6)+3*x(7)-3*x(8)-6*x(9)+3*x(10);
y(7) = -6*x(5)-3*x(6)+3*x(7)+6*x(8)+3*x(9)-3*x(10);
y(8) = 2*x(5)-3*x(6)+x(7)-2*x(8)+3*x(9)-x(10);
y(9) = x(5)-x(7)-x(8)+x(10);
y(10) = -12*x(5)+6*x(6)+12*x(7)+6*x(8)-6*x(9)-12*x(10)+6*x(11);
y(11) = 6*x(4)-3*x(5)-6*x(6)+3*x(7);
y(12) = 6*x(5)+3*x(6)-3*x(7);
y(13) = -2*x(5)+3*x(6)-x(7);
y(14) = -x(5)+x(7);
y(15) = 12*x(5)-6*x(6)-12*x(7)+6*x(8);
y = y/6;
```

8 point linear convolution

Equation:

$$D_8 = D_2 \otimes D_2 \otimes D_2$$

Equation:

$$F_8 = \begin{matrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & -1 & 0 & 1 & -1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\
1 & -1 & 0 & 1 & 1 & 0 & -1 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & -1 & 1 \\
-1 & -1 & 1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & 1 & 1 & -1 & 1 & 1 \\
0 & 1 & 0 & -1 & 1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 & 1 & 0 & 0 & -1 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

```
function y = F8t(x)
y = zeros(27,1);
y(1) = x(1)-x(2)-x(3)+x(4)-x(5)+x(6)+x(7)-x(8);
y(2) = -x(2)+x(3)+x(4)-x(5)+x(6)-x(7)-x(8)+x(9);
y(3) = x(2)-x(4)-x(6)+x(8);
```

```
y(4) = -x(3)+x(4)+x(5)-x(6)+x(7)-x(8)-x(9)+x(10);
y(5) = x(4)-x(5)-x(6)+x(7)-x(8)+x(9)+x(10)-x(11);
y(6) = -x(4)+x(6)+x(8)-x(10);
y(7) = x(3)-x(4)-x(7)+x(8);
y(8) = -x(4)+x(5)+x(8)-x(9);
y(9) = x(4)-x(8);
y(10) = -x(5)+x(6)+x(7)-x(8)+x(9)-x(10)-x(11)+x(12);
y(11) = x(6)-x(7)-x(8)+x(9)-x(10)+x(11)+x(12)-x(13);
y(12) = -x(6)+x(8)+x(10)-x(12);
y(13) = x(7)-x(8)-x(9)+x(10)-x(11)+x(12)+x(13)-x(14);
y(14) = -x(8)+x(9)+x(10)-x(11)+x(12)-x(13)-x(14)+x(15);
y(15) = x(8)-x(10)-x(12)+x(14);
y(16) = -x(7)+x(8)+x(11)-x(12);
y(17) = x(8)-x(9)-x(12)+x(13);
y(18) = -x(8)+x(12);
y(19) = x(5)-x(6)-x(7)+x(8);
y(20) = -x(6)+x(7)+x(8)-x(9);
y(21) = x(6)-x(8);
y(22) = -x(7)+x(8)+x(9)-x(10);
y(23) = x(8)-x(9)-x(10)+x(11);
y(24) = -x(8)+x(10);
y(25) = x(7)-x(8);
y(26) = -x(8)+x(9);
y(27) = x(8);
```

18 point linear convolution

Equation:

$$D_8 = D_2 \otimes D_3 \otimes D_3$$

F_{18} and the program F18t are too big to print.

Appendix: A 45 Point Circular Convolution Program

As an example, we list a 45 point circular convolution program.

```
function y = cconv45(x,u)
% y = ccconv45(x,u)
% y : the 45 point circular convolution of x and h
% where u is a vector of precomputed
multiplicative constants

x = pfp([9,5],2,x);                % prime factor
permuation
x = KRED([3,5],[2,1],2,x);         % reduction
operations (152 Additions)
y = zeros(45,1);
% -------------------- block : 1 -----------------
--------------------------------
y(1) = x(1)*u(1);                  % 1 Multiplication
% -------------------- block : 3 -----------------
--------------------------------
v = ID2I(1,1,x(2:3));              % v = (I(1) kron D2
kron I(1)) * x(2:3)                a : 1=1*1
v = v.*u(2:4);                     % 3 Multiplications
y(2:3) = ID2tI(1,1,v);             % y(2:3) = (I(1)
kron D2' kron I(1)) * v               a : 2=1*2
% -------------------- block : 9 -----------------
--------------------------------
v = ID3I(2,1,x(4:9));              % v = (I(2) kron D3
kron I(1)) * x(4:9)                a : 14=2*7
v = ID2I(1,5,v);                   % v = (I(1) kron D2
kron I(5)) * v                     a : 5=5*1
v = v.*u(5:19);                    % 15 Multiplications
v = ID2tI(1,5,v);                  % v = (I(1) kron D2'
kron I(5)) * v                    a : 10=5*2
y(4:9) = ID3tI(2,1,v);             % y(4:9) = (I(2)
```

```
kron D3' kron I(1)) * v          a : 18=2*9
% -------------------- block : 5 -----------------
-------------------------------
v = ID2I(1,2,x(10:13));       % v = (I(1) kron D2
kron I(2)) * x(10:13)         a : 2=2*1
v = ID2I(3,1,v);              % v = (I(3) kron D2
kron I(1)) * v                a : 3=3*1
v = v.*u(20:28);              % 9 Multiplications
v = ID2tI(1,3,v);             % v = (I(1) kron D2'
kron I(3)) * v               a : 6=3*2
y(10:13) = ID2tI(2,1,v);      % y(10:13) = (I(2)
kron D2' kron I(1)) * v        a : 4=2*2
% -------------------- block : 15 = 3 * 5 --------
-------------------------------
v = ID2I(1,4,x(14:21));       % v = (I(1) kron D2
kron I(4)) * x(14:21)         a : 4=4*1
v = ID2I(3,2,v);              % v = (I(3) kron D2
kron I(2)) * v                a : 6=6*1
v = ID2I(9,1,v);              % v = (I(9) kron D2
kron I(1)) * v                a : 9=9*1
v = v.*u(29:55);              % 27 Multiplications
v = ID2tI(1,9,v);             % v = (I(1) kron D2'
kron I(9)) * v               a : 18=9*2
v = ID2tI(2,3,v);             % v = (I(2) kron D2'
kron I(3)) * v               a : 12=6*2
y(14:21) = ID2tI(4,1,v);      % y(14:21) = (I(4)
kron D2' kron I(1)) * v        a : 8=4*2
% -------------------- block : 45 = 9 * 5 --------
-------------------------------
v = ID3I(2,4,x(22:45));       % v = (I(2) kron D3
kron I(4)) * x(22:45)         a : 56=8*7
v = ID2I(1,20,v);             % v = (I(1) kron D2
kron I(20)) * v               a : 20=20*1
v = ID2I(15,2,v);             % v = (I(15) kron D2
kron I(2)) * v               a : 30=30*1
v = ID2I(45,1,v);             % v = (I(45) kron D2
kron I(1)) * v               a : 45=45*1
```

```
v = v.*u(56:190);              % 135
Multiplications
v = ID2tI(1,45,v);             % v = (I(1) kron D2'
kron I(45)) * v               a : 90=45*2
v = ID2tI(10,3,v);             % v = (I(10) kron
D2' kron I(3)) * v               a : 60=30*2
v = ID2tI(20,1,v);             % v = (I(20) kron
D2' kron I(1)) * v               a : 40=20*2
y(22:45) = ID3tI(2,4,v);       % y(22:45) = (I(2)
kron D3' kron I(4)) * v          a : 72=8*9

y = tKRED([3,5],[2,1],2,y);         % transpose
reduction operations (152 Additions)
y = pfpt([9,5],2,y);                % prime factor
permuation
y = y(45:-1:1);

% Total Number of Multiplications : 190
% Total Number of Additions: 839
```

Appendix: A 31 Point FFT Program

As an example, we list a 31 point FFT program.

```
function y = fft31(x,u,ip,op)
% y = fft31(x,u,ip,op)
% y  : the 31 point DFT of x
% u  : a vector of precomputed multiplicative
constants
% ip : input permutation
% op : ouput permutation

y = zeros(31,1);
x = x(ip);
% input permutation
x(2:31) = KRED([2,3,5],[1,1,1],3,x(2:31));
% reduction operations
y(1) = x(1)+x(2);
% DC term calculation
% -------------------- block : 1 -----------------
--------------------------------
y(2) = x(2)*u(1);
% -------------------- block : 2 -----------------
--------------------------------
y(3) = x(3)*u(2);
% -------------------- block : 3 -----------------
--------------------------------
v = ID2I(1,1,x(4:5));              % v = (I(1)
kron D2 kron I(1)) * x(4:5)
v = v.*u(3:5);
y(4:5) = ID2tI(1,1,v);             % y(4:5) =
(I(1) kron D2' kron I(1)) * v
% -------------------- block : 6 = 2 * 3 ---------
--------------------------------
v = ID2I(1,1,x(6:7));              % v = (I(1)
```

```
kron D2 kron I(1)) * x(6:7)
v = v.*u(6:8);
y(6:7) = ID2tI(1,1,v);              % y(6:7) =
(I(1) kron D2' kron I(1)) * v
% -------------------- block : 5 -----------------
--------------------------------
v = ID2I(1,2,x(8:11));              % v = (I(1)
kron D2 kron I(2)) * x(8:11)
v = ID2I(3,1,v);                    % v = (I(3)
kron D2 kron I(1)) * v
v = v.*u(9:17);
v = ID2tI(1,3,v);                   % v = (I(1)
kron D2' kron I(3)) * v
y(8:11) = ID2tI(2,1,v);             % y(8:11) =
(I(2) kron D2' kron I(1)) * v
% -------------------- block : 10 = 2 * 5 --------
--------------------------------
v = ID2I(1,2,x(12:15));             % v = (I(1)
kron D2 kron I(2)) * x(12:15)
v = ID2I(3,1,v);                    % v = (I(3)
kron D2 kron I(1)) * v
v = v.*u(18:26);
v = ID2tI(1,3,v);                   % v = (I(1)
kron D2' kron I(3)) * v
y(12:15) = ID2tI(2,1,v);            % y(12:15) =
(I(2) kron D2' kron I(1)) * v
% -------------------- block : 15 = 3 * 5 --------
--------------------------------
v = ID2I(1,4,x(16:23));             % v = (I(1)
kron D2 kron I(4)) * x(16:23)
v = ID2I(3,2,v);                    % v = (I(3)
kron D2 kron I(2)) * v
v = ID2I(9,1,v);                    % v = (I(9)
kron D2 kron I(1)) * v
v = v.*u(27:53);
v = ID2tI(1,9,v);                   % v = (I(1)
kron D2' kron I(9)) * v
```

```
v = ID2tI(2,3,v);                    % v = (I(2)
kron D2' kron I(3)) * v
y(16:23) = ID2tI(4,1,v);             % y(16:23) =
(I(4) kron D2' kron I(1)) * v
% -------------------- block : 30 = 2 * 3 * 5 ----
--------------------------------
v = ID2I(1,4,x(24:31));              % v = (I(1)
kron D2 kron I(4)) * x(24:31)
v = ID2I(3,2,v);                     % v = (I(3)
kron D2 kron I(2)) * v
v = ID2I(9,1,v);                     % v = (I(9)
kron D2 kron I(1)) * v
v = v.*u(54:80);
v = ID2tI(1,9,v);                    % v = (I(1)
kron D2' kron I(9)) * v
v = ID2tI(2,3,v);                    % v = (I(2)
kron D2' kron I(3)) * v
y(24:31) = ID2tI(4,1,v);             % y(24:31) =
(I(4) kron D2' kron I(1)) * v
% ----------------------------------------------
--------------------------------
y(2) = y(1)+y(2);
% DC term calculation
y(2:31) = tKRED([2,3,5],[1,1,1],3,y(2:31));
% transpose reduction operations
y = y(op);
% output permutation

% For complex data -
% Total Number of Real Multiplications : 160
% Total Number of Real Additions: 776
```

The constants, input and output permutations are:

```
% The multiplicative constants for the 31 point
FFT
```

```
I = sqrt(-1);
u = [
        -1.033333333333333
         0.185592145427667*I
         0.251026872929094
         0.638094290379888
        -0.296373721102994
        -0.462201919825109*I
         0.155909426230360*I
         0.102097497864916*I
        -0.100498239164838
        -0.217421331841463
        -0.325082164955763
         0.798589508696894
        -0.780994042074251
        -0.256086011899669
         0.169494392220932
         0.711997889018157
        -0.060064820876732
        -1.235197570427205*I
        -0.271691369288525*I
         0.541789612349592*I
         0.329410560797314*I
         1.317497505049809*I
        -0.599508803858381*I
         0.093899154219231*I
        -0.176199088841836*I
         0.028003825226279*I
         1.316699050305790
         1.330315270540553
        -0.385122753006171
        -2.958666546021397
        -2.535301995146201
         2.013474028487015
         1.081897731187396
         0.136705213653014
```

-0.569390844064251
-0.262247009112805
 2.009855570455675
-1.159348599757857
 0.629367699727360
 1.229312102919654
-1.479874670425178
-0.058279061554516
-0.908786032252333
 0.721257672797977
-0.351484013730995
-1.113390280332076
 0.514823784254676
 0.776432948764679
 0.435329964075516
-0.177866452687279
-0.341206223210960
 0.257360272866440
-0.050622276244575
-2.745673340229639*I
 2.685177424507523*I
 0.880463026400118*I
-5.028851220636894*I
-0.345528375980267*I
 1.463210769729252*I
 3.328421083558774*I
-0.237219367348867*I
-1.086975102467855*I
-1.665522956385442*I
 1.628826188810638*I
 0.534088072762272*I
-3.050496586573981*I
-0.209597199290132*I
 0.887582325001072*I
 2.019017208624242*I
-0.143897052948668*I
-0.659358110687783*I

```
         1.470398765538361*I
        -1.438001204439387*I
        -0.471517033054130*I
         2.693115935736959*I
         0.185041858423467*I
        -0.783597698243441*I
        -1.782479430727672*I
         0.127038806765845*I
         0.582111071051880*I
];


% The input permutation for the 31 point FFT

ip = [
   1
   2
   17
   9
   5
   3
   26
   29
   15
   8
   20
   6
   19
   10
   21
   11
   31
   16
   24
   28
   30
   7
```

```
   4
   18
   25
   13
   27
   14
   23
   12
   22
];


% The output permutation for the 31 point FFT

op = [
   1
   31
   30
   2
   29
   26
   6
   19
   28
   23
   25
   9
   5
   7
   18
   12
   27
   3
   22
   20
   24
   10
```

```
    8
    13
    4
    21
    11
    14
    17
    15
    16
];
```

Appendix: Matlab Functions For Circular Convolution and Prime Length FFTs

Programs for Reduction Operations

The reduction matrix of Equation 44 in Preliminaries is implemented by `KRED` which calls `RED` . Its transpose and inverse transpose are implemented by `tRED` , `tRED` , `itKRED` and `itRED` .

```
function x = KRED(P,E,K,x)
% x = KRED(P,E,K,x);
% P : P = [P(1),...,P(K)];
% E : E = [E(K),...,E(K)];
for i = 1:K
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
   e = E(i);
   for j = e-1:-1:0
      x(1:a*c*(p^(j+1))) = RED(p,a,c*
(p^j),x(1:a*c*(p^(j+1))));
   end
end

function y = RED(p,a,c,x)
% y = RED(p,a,c,x);
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      y(i+j+1) = x(i*p+j+1);
      for k = 0:c:c*(p-2)
         y(i+j+1) = y(i+j+1) + x(i*p+j+k+c+1);
         y(i*(p-1)+j+k+a*c+1) = x(i*p+j+k+1) -
x(i*p+j+c*(p-1)+1);
      end
   end
end
```

```
function x = tKRED(P,E,K,x)
% x = tKRED(P,E,K,x);
% (transpose)
% P : P = [P(1),...,P(K)];
% E : E = [E(K),...,E(K)];
for i = K:-1:1
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
   e = E(i);
   for j = 0:e-1
      x(1:a*c*(p^(j+1))) = tRED(p,a,c*
(p^j),x(1:a*c*(p^(j+1))));
   end
end

function y = tRED(p,a,c,x)
% y = tRED(p,a,c,x);
% (transpose)
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      y(i*p+j+c*(p-1)+1) = x(i+j+1);
      for k = 0:c:c*(p-2)
         y(i*p+j+k+1) = x(i+j+1) + x(i*(p-
1)+j+k+a*c+1);
         y(i*p+j+c*(p-1)+1) = y(i*p+j+c*(p-1)+1) -
x(i*(p-1)+j+k+a*c+1);
      end
   end
end
```

Programs for I ⊗ Dk ⊗ I

The operations of $I_m \otimes D_2 \otimes I_n$ and $I_m \otimes D_3 \otimes I_n$ are carried out by `ID2I` and `ID3I` . Their transposes by `ID2tI` and `ID3tI` . The functions `D2` and `D3` are listed in the appendix, `Bilinear Forms for Linear

Convolution.' Two versions of `ID2I` are listed here. One of them calls `D2` in a loop, while the other version puts the `D2` code in the loop instead of using a function call. There are several ways to implement the form $I \otimes D_2 \otimes I$. But this is a simple and straightforward method. It is modeled after `IAI` in the text.

```
function y = ID2I(m,n,x)
y = zeros(m*n*3,1);
v = 0:n:2*n;
u = 0:n:n;
for i = 0:m-1
   for j = 0:n-1
      y(v+i*3*n+j+1) = D2(x(u+i*2*n+j+1));
   end
end
```

```
function y = ID2I(m,n,x)
y = zeros(m*n*3,1);
for i = 0:n:n*(m-1)
   i2 = 2*i;
   i3 = 3*i;
   for j = 1:n
      j2 = i2 + j;
      j3 = i3 + j;
      y(j3) = x(j2);
      y(n+j3) = x(n+j2);
      y(2*n+j3) = x(j2) + x(n+j2);
   end
end
```

```
function y = ID2tI(m,n,x)
y = zeros(m*n*2,1);
v = 0:n:n;
u = 0:n:2*n;
for i = 0:m-1
   for j = 0:n-1
      y(v+i*2*n+j+1) = D2t(x(u+i*3*n+j+1));
```

```
   end
end

function y = ID3I(m,n,x)
y = zeros(m*n*5,1);
v = 0:n:4*n;
u = 0:n:2*n;
for i = 0:m-1
   for j = 0:n-1
      y(v+i*5*n+j+1) = D3(x(u+i*3*n+j+1));
   end
end

function y = ID3tI(m,n,x)
y = zeros(m*n*3,1);
v = 0:n:2*n;
u = 0:n:4*n;
for i = 0:m-1
   for j = 0:n-1
      y(v+i*3*n+j+1) = D3t(x(u+i*5*n+j+1));
   end
end
```

Appendix: A Matlab Program for Generating Prime Length FFT Programs

```
 function [u,ip,op,ADDS,MULTS] = ff(p,e);
% [u,ip,op,ADDS,MULTS] = ff(p,e);
% u  : multiplicative constants
% ip : input permutation
% op : output permutation

K = length(p);
N = prod(p.^e);
P = N + 1;
[pr, ipr] = primitive_root(P);
Red_Adds = 2 * N * (K - sum(1./(p.^e)) );
ADDS = 2 * Red_Adds;

FS = sprintf('fft%d.m',P);
fid = fopen(FS,'w');

fprintf(fid,'function y = fft%d(x,u,ip,op)\n',P);
fprintf(fid,'%% y = fft%d(x,u,ip,op)\n',P);
fprintf(fid,'%% y  : the %d point DFT of x \n',P);
fprintf(fid,'%% u  : a vector of precomputed
multiplicative constants\n');
fprintf(fid,'%% ip : input permutation\n');
fprintf(fid,'%% op : ouput permutation\n');

Pstr = sprintf('[%d',p(1));
for k = 2:K, Pstr = [Pstr, sprintf(',%d',p(k))];
end
Pstr = [Pstr,']'];
Estr = sprintf('[%d',e(1));
for k = 2:K, Estr = [Estr, sprintf(',%d',e(k))];
end
Estr = [Estr,']'];
PEstr = sprintf('[%d',p(1)^e(1));
for k = 2:K, PEstr = [PEstr,
```

```
sprintf(',%d',p(k)^e(k))]; end
PEstr = [PEstr,']'];

fprintf(fid,'\n');
S = sprintf('y = zeros(%d,1);\n',P);
fprintf(fid,S);
S1 = sprintf('x = x(ip);');
S2 = sprintf('%% input permutation\n');
fprintf(fid,'%-50s%s',S1,S2);
S1 = sprintf(['x(2:%d) =
KRED(',Pstr,',',Estr,',%d,x(2:%d));'],P,K,P);
S2 = sprintf('%% reduction operations\n');
fprintf(fid,'%-50s%s',S1,S2);

e_table = [0:e(1)]';
a = e(1)+1;
for i = 2:K
   e_table = [kron(ones(e(i)+1,1),e_table),
kron([0:e(i)]',ones(a,1))];
   a = a * (e(i)+1);
end
R = prod(e+1);


% ------------------------ MULTIPLICATIVE
CONSTANTS ------------------------
k = rp(P,ipr,0:N);
I = sqrt(-1);
W = exp(-I*2*pi*k/P);
h = W(2:P);
h = h(N:-1:1);
h = pfp(p.^e,K,h);
h = itKRED(p,e,K,h);
u = h(1);

S1 = sprintf('y(1) = x(1)+x(2);');
S2 = sprintf('%% DC term calculation\n');
```

```
fprintf(fid,'%-50s%s',S1,S2);

DC_ADDS = 2;
ADDS = ADDS + DC_ADDS;

SLINE = '----------------------------------------
----------------------------------';
SB = ' block : 1 ';
SC = SLINE;
BL = 21;
SC(BL:BL-1+length(SB)) = SB;
fprintf(fid,'%% %s\n',SC);
S1 = sprintf('y(2) = x(2)*u(1);');
fprintf(fid,'%-40s\n',S1);
a = 1;
MULTS = 1;
for i = 2:R
   v = e_table(i,:);
   f = find(v>0);
   q = p(f);
   t = v(f);
   L = prod(q-1)*prod(q.^(t-1));

   B = prod(q.^t);
   bs = sprintf('%d',q(1)^t(1));
   for k = 2:length(q), bs = [bs, sprintf(' *
%d',q(k)^t(k))]; end
   if length(q) > 1
        SB = sprintf(' block : %d = %s ',B,bs);
        SC = SLINE;
        SC(BL:BL-1+length(SB)) = SB;
        fprintf(fid,'%% %s\n',SC);
   else
        SB = sprintf(' block : %d ',B);
        SC = SLINE;
        SC(BL:BL-1+length(SB)) = SB;
        fprintf(fid,'%% %s\n',SC);
```

```
   end
   if prod(q.^t) == 2
      S1 = sprintf('y(%d) =
x(%d)*u(%d);',a+2,a+2,MULTS+1);
      fprintf(fid,'%-40s\n',S1);
      Mk = 1;
   else
      d = []; r = []; c = []; Q = []; Qt = [];
      for j = 1:length(q)
         [dk,rk,ck,Qk,Qtk] = A_data(q(j)^t(j));
         if dk > 1
            d = [d dk]; r = [r rk]; c = [c ck]; Q
= [Q Qk]; Qt = [Qt Qtk];
         end
      end
      [g,C1] = cgc(Q,r,c,length(Q));
      ADDS = ADDS + C1;
      Mk = prod(r);
      BEG = int2str(a+2); FIN = int2str(a+1+L);
      XX = ['x(',BEG,':',FIN,')']; YY = 'v';
      kpi(d,g,r,c,length(Q),YY,XX,fid);
      S1 = ['v =
v.*u(',int2str(MULTS+1),':',int2str(MULTS+Mk),');'
];
      fprintf(fid,'%-40s\n',S1);
      [g,C2] = cgc(Qt,c,r,length(Q));
      ADDS = ADDS + C2;
      XX = 'v'; YY = ['y(',BEG,':',FIN,')'];
      kpit(d,g,c,r,length(Q),YY,XX,fid);
   end

   c = [];
   r = [];
   lq = length(q);
   for j = 1:lq
      [fk,rk,ck] = C_data(q(j),t(j));
      r = [r rk]; c = [c ck];
```

```
   end
   f = (q-1).*(q.^(t-1));
   temp = Kcrot(q,t,lq,h(a+1:a+L));
   temp = KFt(f,r,c,temp);
   u = [u; temp(:)];
   a = a + L;
   MULTS = MULTS + Mk;
end

u(1) = u(1)-1;
fprintf(fid,'%% %s\n',SLINE);
S1 = sprintf('y(2) = y(1)+y(2);');
S2 = sprintf('%% DC term calculation\n');
fprintf(fid,'%-50s%s',S1,S2);
S1 = sprintf(['y(2:%d) =
tKRED(',Pstr,',',Estr,',%d,y(2:%d));'],P,K,P);
S2 = sprintf('%% transpose reduction
operations\n');
fprintf(fid,'%-50s%s',S1,S2);
S1 = sprintf('y = y(op);');
S2 = sprintf('%% output permutation\n');
fprintf(fid,'%-50s%s',S1,S2);
fprintf(fid,'\n');

MULTS = 2 * MULTS;
ADDS = 2* ADDS;
fprintf(fid,'%% For complex data - \n');
fprintf(fid,'%% Total Number of Real
Multiplications : %d\n',MULTS);
fprintf(fid,'%% Total Number of Real Additions:
%d\n\n',ADDS);
fclose(fid);

%%%%%%%%%%%%%%%%%%% COMPUTE INPUT AND OUTPUT
PERMUTATIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

id = 1:P;   % identity permutation
```

```
ip = rp(P,pr,id);
ip(2:P) = pfp(p.^e,K,ip(2:P));

op = id;
op(2:P) = pfpt(p.^e,K,op(2:P));
op(2:P) = op(P:-1:2);
op = rpt(P,ipr,op);

%%%%%%%%%%%%%% PUT MULTIPLICATIVE CONSTANTS AND
PERMUTATIONS IN A FILE %%%%%%%%%%%%%

CFS = sprintf('cap%d.m',P);
fid = fopen(CFS,'w');
fprintf(fid,'\n%% The multiplicative constants for
the %d point FFT\n\n',P);
fprintf(fid,'I = sqrt(-1);\n');
fprintf(fid,'u = [\n');
for k = 1:MULTS/2
   if abs(real(u(k))) < 0.000001
      fprintf(fid,'%25.15f*I\n',imag(u(k)));
   elseif abs(imag(u(k))) < 0.00001
      fprintf(fid,'%25.15f\n',real(u(k)));
   else
      fprintf(fid,'%25.15f +
%25.15f*I\n',real(u(k)),imag(u(k)));
   end
end
fprintf(fid,'];\n\n');
fprintf(fid,'\n%% The input permutation for the %d
point FFT\n\n',P);
fprintf(fid,'ip = [\n');
for k = 1:P
        fprintf(fid,'   %d\n',ip(k));
end
fprintf(fid,'];\n\n');
fprintf(fid,'\n%% The output permutation for the
%d point FFT\n\n',P);
```

```
fprintf(fid,'op = [\n');
for k = 1:P
        fprintf(fid,'   %d\n',op(k));
end
fprintf(fid,'];\n\n');
fclose(fid);
```

The following programs print the program statements that carry out the operation $I \otimes D_k \otimes I$ and $I \otimes D_k^t \otimes I$. They are modeled after `kpi` in the text.

```
function kpi(d,g,r,c,n,Y,X,fid)
% kpi(d,g,r,c,n,Y,X,fid);
% Kronecker Product : A(d(1)) kron ... kron
A(d(n))
% g : permutation of 1,...,n
% r : [r(1),...,r(n)]
% c : [c(1),..,c(n)]
% r(i) : rows of A(d(i))
% c(i) : columns of A(d(i))
% n : number of terms
for i = 1:n
   a = 1;
   for k = 1:(g(i)-1)
      if i > find(g==k)
         a = a * r(k);
      else
         a = a * c(k);
      end
   end
   b = 1;
   for k = (g(i)+1):n
      if i > find(g==k)
         b = b * r(k);
      else
         b = b * c(k);
      end
```

```
   end
   % Y = (I(a) kron A(d(g(i))) kron I(b)) * X;
   if i == 1
      S1 = sprintf([Y,' = ID%dI(%d,%d,',X,');
'],d(g(i)),a,b);
      S2 = sprintf(['%% ',Y,' = (I(%d) kron D%d
kron I(%d)) * ',X],a,d(g(i)),b);
      fprintf(fid,'%-35s%s\n',S1,S2);
   elseif d(g(i)) ~= 1
      S1 = sprintf([Y,' = ID%dI(%d,%d,',Y,');
'],d(g(i)),a,b);
      S2 = sprintf(['%% ',Y,' = (I(%d) kron D%d
kron I(%d)) * ',Y],a,d(g(i)),b);
      fprintf(fid,'%-35s%s\n',S1,S2);
   end
end

function kpit(d,g,r,c,n,Y,X,fid)
% kpit(g,r,c,n,Y,X,fid);
% (transpose)
% Kronecker Product : A(d(1))' kron ... kron
A(d(n))'
% g : permutation of 1,...,n
% r : [r(1),...,r(n)]
% c : [c(1),..,c(n)]
% r(i) : rows of A(d(i))'
% c(i) : columns of A(d(i))'
% n : number of terms

for i = 1:n
   a = 1;
   for k = 1:(g(i)-1)
      if i > find(g==k)
         a = a * r(k);
      else
         a = a * c(k);
      end
```

```
   end
   b = 1;
   for k = (g(i)+1):n
      if i > find(g==k)
         b = b * r(k);
      else
         b = b * c(k);
      end
   end
   % x = (I(a) kron A(d(g(i)))'' kron I(b)) * x;
   if i == n
      S1 = sprintf([Y,' = ID%dtI(%d,%d,',X,');
'],d(g(i)),a,b);
      S2 = sprintf(['%% ',Y,' = (I(%d) kron D%d''
kron I(%d)) * ',X],a,d(g(i)),b);
      fprintf(fid,'%-35s%s\n',S1,S2);
   elseif d(g(i)) ~= 1
      S1 = sprintf([X,' = ID%dtI(%d,%d,',X,');
'],d(g(i)),a,b);
      S2 = sprintf(['%% ',X,' = (I(%d) kron D%d''
kron I(%d)) * ',X],a,d(g(i)),b);
      fprintf(fid,'%-35s%s\n',S1,S2);
   end
end
```

Programs for Computing Multiplicative Constants

The following programs carry out the operation of $F_{d_1} \otimes \cdots \otimes F_{d_K}$ where F is the reconstruction matrix in a linear convolution algorithm. See the appendix, `Bilinear Forms for Linear Convolution.'

```
function u = KFt(f,r,c,u)
% u = (F^t kron ... kron F^t)*u
% (transpose)
% f = [f(1),...,f(K)]
% r : r(i) = rows of F(i)
% c : c(i) = columns of F(i)
```

```
% u : length(u) = prod(c);
K = length(f);
for i = 1:K
   m = prod(c(1:i-1));
   n = prod(r(i+1:K));
   u = IFtI(f(i),r(i),c(i),m,n,u);
end

function y = IFtI(s,r,c,m,n,x);
% y = (I(m) kron F(s)^t kron I(n))*x
% (transpose)
% r : rows of F(s)
% c : columns of F(s)
v = 0:n:n*(c-1);
u = 0:n:n*(r-1);
for i = 0:m-1
   for j = 0:n-1
      y(v+i*c*n+j+1) = Ftop(s,x(u+i*r*n+j+1));
   end
end

function y = Ftop(k,x)
if k == 1, y = x;
elseif k == 2, y = F2t(x);
elseif k == 3, y = F3t(x);
elseif k == 4, y = F4t(x);
elseif k == 6, y = F6t(x);
elseif k == 8, y = F8t(x);
elseif k == 18, y = F18t(x);
end
```

The following programs carry out the operation of $G_{p_1^{e_1}} \otimes \cdots \otimes G_{p_K^{e_K}}$ were G is given by Equation 13 and Equation 14 from Bilinear Forms for Circular Convolution.

```
function x = Kcrot(p,e,K,x)
% Kronecker product of Cyclotomic Reduction
```

```
Operations.
% x = (G(p(1)^e(1)) kron ... kron G(p(K)^(K)))^t*x
% (transpose)
% p : p = [p(1),...,p(K)];
% e : e = [e(1),...,e(K)];
a = (p-1).*((p).^(e-1));
r = a;           % r(i) = number of rows of G(i)
c = 2*a-1;       % c(i) = number of columns of G(i)
m = 1;
n = prod(r);
for i = 1:K
   n = n / r(i);
   x = IcrotI(p(i),e(i),m,n,x);
   m = m * c(i);
end

function y = IcrotI(p,e,m,n,x)
%  y = (eye(m) kron G(p^e)^t kron eye(n))*x
%  (transpose)
a = (p-1)*(p^(e-1));
c = a;
r = 2*a-1;
y = zeros(r*m*n,1);
v = 0:n:(r-1)*n;
u = 0:n:(c-1)*n;
for i = 0:m-1
   for j = 0:n-1
      y(v+i*r*n+j+1) = crot(p,e,x(u+i*c*n+j+1));
   end
end

function y = crot(p,e,x)
% y = crot(p,x)
% cyclotomic reduction matrix (transpose)
% length(x) == 2*n-1
% length(y) == n
% where n = (p-1)*(p^(e-1))
```

```
n = (p-1)*(p^(e-1));
y = zeros(2*n-1,1);
if p == 2
  n = p^(e-1);
  y(1:n) = x;
  y(n+1:2*n-1) = -x(1:n-1);
else
  y(1:n) = x;
  L = p^(e-1);
  y(n+1:n+L) = -x(1:L);
  a = L;
  for k = 2:p-1
     y(n+1:n+L) = y(n+1:n+L) - x(a+1:a+L);
     a = a + L;
  end
  b = 2*n-1 - p*(p^(e-1));
  y(p*L+1:p*L+b) = x(1:b);
end
```

The following programs tell the programs for code generation relevant information about the bilinear forms for cyclotomic convolution. Specifically, they indicates the linear convolution out of which these cyclotomic convolution are composed, and the dimensions of the corresponding matrices. See the appendix Bilinear Forms for Linear Convolution.

```
function [d,r,c,Q,Qt] = A_data(n)
% A : A matrix in bilinear form for cyclotomic
convolution
% d : linear convolution modules used
% r : rows
% c : columns
% Q : Q(i) = cost associated with D(d(i))
% Qt : Qt(i) = cost associated with D(d(i))'
if n == 2, d = [1];
elseif n == 4, d = [2];
elseif n == 8, d = [2 2];
```

```
elseif n == 16, d = [2 2 2];
elseif n == 3, d = [2];
elseif n == 9, d = [2 3];
elseif n == 27, d = [2 3 3];
elseif n == 5, d = [2 2];
elseif n == 7, d = [2 3];
end
r = []; c = []; Q = []; Qt = [];
for k = 1:length(d)
   [rk, ck, Qk, Qtk] = D_data(d(k));
   r = [r rk]; c = [c ck]; Q = [Q Qk]; Qt = [Qt
Qtk];
end

function [r,c,Q,Qt] = D_data(d);
% D : D matrix in bilinear form for linear
convolution
% r : rows
% c : columns
% Q : cost associated with D(d)
% Qt : cost associated with D(d)'
if d == 1, r = 1; c = 1; Q = 0; Qt = 0;
elseif d == 2, r = 3; c = 2; Q = 1; Qt = 2;
elseif d == 3, r = 5; c = 3; Q = 7; Qt = 9;
end

function [f,r,c] = C_data(p,e)
% f : length of linear convolution
% r : rows
% c : columns
f = prod((p-1).*(p.^(e-1)));
% (Euler Totient Function)
r = 2*f-1;
c = F_data(f);

function c = F_data(n)
% c : columns of F matrix
```

```
if n == 1, c = 1;
elseif n == 2, c = 3;
elseif n == 4, c = 9;
elseif n == 8, c = 27;
elseif n == 3, c = 5;
elseif n == 6, c = 15;
elseif n == 18, c = 75;
end
```

Programs for Inverse Transpose Reduction Operations

```
function x = itKRED(P,E,K,x)
% x = itKRED(P,E,K,x);
% (inverse transpose)
% P : P = [P(1),...,P(K)];
% E : E = [E(K),...,E(K)];
for i = 1:K
   a = prod(P(1:i-1).^E(1:i-1));
   c = prod(P(i+1:K).^E(i+1:K));
   p = P(i);
   e = E(i);
   for j = e-1:-1:0
      x(1:a*c*(p^(j+1))) = itRED(p,a,c*
(p^j),x(1:a*c*(p^(j+1))));
   end
end

function y = itRED(p,a,c,x)
% y = itRED(p,a,c,x);
% (inverse transpose)
y = zeros(a*c*p,1);
for i = 0:c:(a-1)*c
   for j = 0:c-1
      A = x(i*p+j+1);
      for k = 0:c:c*(p-2)
         A = A + x(i*p+j+k+c+1);
```

```
      end
      y(i+j+1) = A;
      for k = 0:c:c*(p-2)
         y(i*(p-1)+j+k+a*c+1) = p*x(i*p+j+k+1) -
A;
      end
   end
end
y = y/p;
```

Programs for Permutations

The permutation of Equation 18 from Preliminaries is implemented by `pfp` . It calls the function `pfp2I` . The transpose is implemented by `pfpt` and it calls `pfpt2I` .

```
function x = pfp(n,K,x)
% x = P(n(1),...,n(K)) * x
% n = [n(1),...,n(K)];
% length(x) = prod(n(1),...,n(K))
a = prod(n);
s = 1;
for i = K:-1:2
  a = a / n(i);
  x = pfp2I(a,n(i),s,x);
  s = s * n(i);
end

function y = pfp2I(a,b,s,x)
% y = kron(P(a,b),I(s)) * x;
% length(x) = a*b*s
n = a * b;
y = zeros(n*s,1);
k1 = 0;
k2 = 0;
for k = 0:n-1
  i1 = s * (k1 + b * k2);
```

```
  i2 = s * k;
  for i = 1:s
    y(i1 + i) = x(i2 + i);
  end
  k1 = k1 + 1;
  k2 = k2 + 1;
  if k1 >= b
    k1 = k1 - b;
  end
  if k2 >= a
    k2 = k2 - a;
  end
end

function x = pfpt(n,K,x)
% x = P(n(1),...,n(K))' * x
% (tanspose)
% n = [n(1),...,n(K)];
% length(x) = prod(n(1),...,n(K))
% a = prod(n);
a = n(1);
s = prod(n(2:K));
for i = 2:K
  s = s / n(i);
  x = pfpt2I(a,n(i),s,x);
  a = a * n(i);
end

function y = pfpt2I(a,b,s,x)
% y = P(a,b)' kron I(s) * x;
% (transpose)
% length(x) = a*b*s
n = a * b;
y = zeros(n*s,1);
k1 = 0;
k2 = 0;
for k = 0:n-1
```

```
  i1 = s * (k1 + b * k2);
  i2 = s * k;
  for i = 1:s
    y(i2 + i) = x(i1 + i);
  end
  k1 = k1 + 1;
  k2 = k2 + 1;
  if k1 >= b
    k1 = k1 - b;
  end
  if k2 >= a
    k2 = k2 - a;
  end
end
```

The following Matlab programs implement Rader's permutation and its transpose. They require the primitive root to be passed to them as an argument.

```
function y = rp(p,r,x)
% Rader's Permutation
% p : prime
% r : a primitive root of p
% x : length(x) == p
a = 1;
y = zeros(p,1);
y(1) = x(1);
for k = 2:p
   y(k) = x(a+1);
   a = rem(a*r,p);
end

function y = rpt(p,r,x)
% Rader's Permutation
% (transpose)
% p : prime
% r : a primitive root of p
```

```
% x : length(x) == p
a = 1;
y = zeros(p,1);
y(1) = x(1);
for k = 2:p
   y(a+1) = x(k);
   a = rem(a*r,p);
end

function [R, R_inv] = primitive_root(N)

%  function [R, R_inv] = primitive_root(N)
%  Ivan Selesnick
%  N is assumed to be prime.  This function
returns R,
%  the smallest primitive root of N, and R_inv,
the
%  inverse of R modulo N.

R = 'Not Found';
m = 0:(N-2);
for x = 1:(N-1)
   if ( 1:(N-1) == sort(rem2(x,m,N)) )
      R = x;
      break
   end
end
R_inv = 'Not Found';
for x = 1:N
   if rem(x*R,N) == 1
      R_inv = x;
      break
   end
end
```